

CrasyDSE: A framework for solving Dyson-Schwinger equations

Markus Q. Huber^a, Mario Mitter^b

^a*Institut für Kernphysik, Technische Universität Darmstadt, Schlossgartenstr. 2, 64289 Darmstadt, Germany*

^b*Institut für Physik, Karl-Franzens-Universität Graz, Universitätsplatz 5, 8010 Graz, Austria*

Abstract

Dyson-Schwinger equations are important tools for non-perturbative analyses of quantum field theories. For example, they are very useful for investigations in quantum chromodynamics and related theories. However, sometimes progress is impeded by the complexity of the equations. Thus automatizing parts of the calculations will certainly be helpful in future investigations. In this article we present a framework for such an automatization based on a *C++* code that can deal with a large number of Green functions. Since also the creation of the expressions for the integrals of the Dyson-Schwinger equations needs to be automatized, we defer this task to a *Mathematica* notebook. We illustrate the complete workflow with an example from Yang-Mills theory coupled to a fundamental scalar field that has been investigated recently. As a second example we calculate the propagators of pure Yang-Mills theory. Our code can serve as a basis for many further investigations where the equations are too complicated to tackle by hand. It also can easily be combined with *DoFun*, a program for the derivation of Dyson-Schwinger equations.¹

Keywords: Dyson-Schwinger equations, correlation functions, quantum field theory

PROGRAM SUMMARY

Program Title: CrasyDSE

Version: 1.1.0

Licensing provisions: CPC non-profit use license

Programming language: *Mathematica* 8 and higher, *C++*

Operating system: all on which *Mathematica* and *C++* are available (Windows, Unix, Mac OS)

PACS: 11.10.-z,03.70.+k,11.15.Tk

CPC Classification: 11.1 General, High Energy Physics and Computing

 11.4 Quantum Electrodynamics

 11.5 Quantum Chromodynamics, Lattice Gauge Theory

 11.6 Phenomenological and Empirical Models and Theories

Nature of problem: Solve (large) systems of Dyson-Schwinger equations numerically.

Solution method: Create *C++* functions in *Mathematica* to be used for the numeric code

Email addresses: markus.huber@physik.tu-darmstadt.de (Markus Q. Huber), mario.mitter@uni-graz.de (Mario Mitter)

¹The code of the program *CrasyDSE* is available at <http://theorie.ikp.physik.tu-darmstadt.de/~mqh/CrasyDSE/>

in *C++*. This code uses structures to handle large numbers of Green functions.

Unusual features: Provides a tool to convert *Mathematica* expressions into *C++* expressions including conversion of function names.

Running time: Depending on the complexity of the investigated system solving the equations numerically can take seconds on a desktop PC to hours on a cluster.

1. Introduction

Strongly coupled field theories play an essential role in the physical description of nature. Both established theories like quantum chromodynamics and conjectured ones like technicolor theories cannot be fully understood without non-perturbative methods. Typical approaches include Monte-Carlo simulations on a discretized space-time or functional equations. Functional renormalization group equations, see, e. g., [1–4], Dyson-Schwinger equations, see, e. g., [5–8], and the n-PI formalism, see, e. g., [9], belong to the second group. Their advantages are well appreciated and they provided many new insights.

In this article we will focus on Dyson-Schwinger equations (DSEs) and propose a concrete way to handle them when they become too complex to be treated by hand alone. DSEs consist of a system of coupled integral equations which relate different Green functions. Since there are infinitely many Green functions there are also infinitely many DSEs. Unfortunately no subset of these equations forms a closed system so that we have to deal with an infinitely large system of equations. Naturally one hopes that only a (small) finite number of Green functions is relevant and looks for truncations capturing the most important features of the theory. Of course, in order to check the validity of such an approach one should test the influence of neglected Green functions. However, this is often very tedious. On the other hand there are also theories where it is known that current truncation and approximation schemes and available methods are insufficient and need to be extended. For example, standard truncations restrict the DSEs to one-loop diagrams [5, 7, 10–12] but Yang-Mills theories in the maximally Abelian gauge require the inclusion of two-loop diagrams in order to be consistent in the non-perturbative regime [13]. Consequently the present technical methods have to be improved.

The reason why more sophisticated truncations or more complicated theories require so much more effort is mainly that the length and complexity of the explicit expressions increase considerably with the number of interactions and the number of external legs. Also the numbers of dressing functions and diagrams grow for higher Green functions. We will illustrate this below explicitly with the example of Yang-Mills theory coupled to a scalar: We will see that extending a simple truncation beyond the propagators by dynamically including the vertex between the gauge field and the scalar triples the number of dressing functions to be calculated and requires five times as many loop integrations. Furthermore, the corresponding integration kernels are substantially more complicated than the first one. Seeing such complexity arise from such a simple extension we felt it was time to think about automatizing this process. This seems even more necessary since computing time is no longer as restrictive as it was ten years ago. For example, fourteen years ago the first solution of the DSE system of Yang-Mills theory that was complete at the propagator level

[10, 14, 15] relied on an angle approximation and took several hours. Nowadays it is possible to do it with the full momentum integration in a few minutes. Thus more complicated truncations and theories are definitely doable. However, right now one has to invest much time in deriving DSEs and implementing them. In a sense we fell behind the possibilities today's computers offer and we think we should try to change this and find means that allow us to focus more on the physical rather than the technical problems.

The technical part of investigating a theory numerically with DSEs consists of two main steps: First, the equations have to be derived. Second, one has to implement them in a numeric code. A tool that assists in the first part is already available with the *Mathematica* [16] application *DoFun* [17]. Here we present a generic numeric code that can serve as a basis for the second step. *CrasyDSE* (Computation of Rather lArge SYstems of DSEs) is capable of dealing with a high number of Green functions and their dressing functions. Furthermore it provides several pre-defined integration routines and numerical approximation techniques. It can also be extended to multi-core environments (see comments in section 4.2) and finite temperature (see comments in section 4.1). Part of *CrasyDSE* is the *Mathematica* package *CrasyDSE.m* to generate *C++* expressions for the kernels. This first of all alleviates the generation of the code tremendously and reduces human errors and secondly is the easiest way to transform the notation of the user into the notation of *CrasyDSE*. Note that the functions of the package can deal with all regular *Mathematica* input and do not rely on *DoFun*. In order to use the package, the file *CrasyDSE.m* has to be copied from `main_Mathematica` to a place where *Mathematica* can find it. We suggest to copy it to the subdirectory `Applications` of the *Mathematica* user directory (`$UserBaseDirectory/Applications`)². Now the package can be loaded with `<<CrasyDSE``.

In the following we will describe the general procedure to solve DSEs in section 2. The numerical problem is formulated in section 3 and section 4 contains details on the provided routines to solve DSEs. Secs. 5 and 6 explain the application of *CrasyDSE* using as examples the calculation of two DSEs of a scalar field coupled to Yang-Mills theory and the calculation of the propagators of pure Yang-Mills theory. Finally, we give a summary and an outlook in section 7. In three appendices we provide details and summaries on functions and variables of *CrasyDSE*.

2. Solving Dyson-Schwinger equations

Our approach to solving DSEs can be separated into three parts as illustrated in Fig. 1:

1. Derive the equations from the given action by the method of choice. If not done in *Mathematica*, enter them into *Mathematica*.
2. Use the *Mathematica* package *CrasyDSE.m* to generate the *C++* files with the kernels. Alternatively, in simple cases one can write the kernel files manually.
3. Use the kernel files with the *C++* code of *CrasyDSE* to solve the DSE numerically.

We illustrate the last steps with two examples in sections 5 and 6.

²On a Unix machine this is typically `~/.Mathematica/Applications`, on Windows it is `User\Application Data\Mathematica\Applications`.

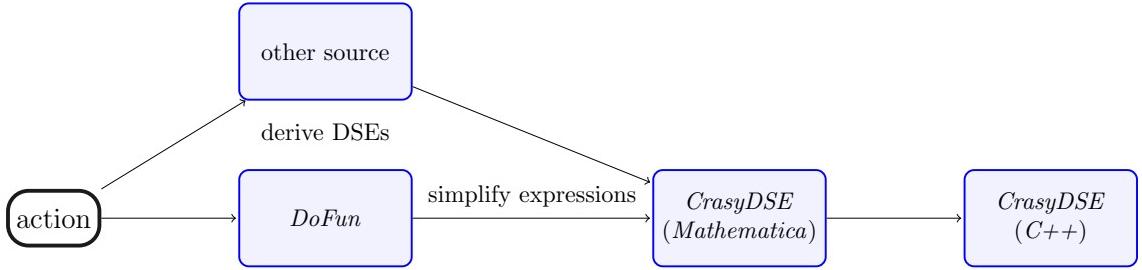


Figure 1: Schematic workflow.

For the first step, the derivation of the DSEs, we recommend the *Mathematica* application *DoFun* (Derivation Of FUNctional equations) [17]. Its predecessor is the *Mathematica* package *DoDSE* (Derivation Of DSEs) [6, 18] which was of great help in the investigation of big systems of DSEs like that of the maximally Abelian gauge [13] and the Gribov-Zwanziger action [19, 20]. The calculation of some infrared properties in the maximally Abelian gauge would even have been impossible without automatization due to the huge number of terms [18, 21]. Later on *DoDSE* was considerably extended and the derivation of functional renormalization group equations³ was included [17], thus it was renamed to *DoFun*. However, note that *CrasyDSE* works completely independent of *DoFun*.

The second step consists in making the *Mathematica* expressions accessible for *C++*. We chose to write our own functions that generate complete *C++* files. Thus we maintain as much control as possible over the process and make it more transparent for the user. However, in principle it would also be possible to let *Mathematica* and *C++* interact in a more direct way via *MathLink*. All the necessary functions to create the *C++* files are included in the package *CrasyDSE.m*, but the notebook from which it is created is also provided so that the user has direct access and can most easily adapt code if required.

Finally, after all kernels have been written into *C++* files, one uses the provided *C++* modules to solve the equations. Typical initial work includes defining model parameters, defining the required Green functions and their dressing functions, choosing integration routines and defining the renormalization procedures. The way dressing functions are defined is thereby quite arbitrary: One can use closed expressions, for example, from fits to lattice data, interpolations or expansions in sets of polynomials. In order to help with starting calculations with *CrasyDSE* we provide several examples with the main code.

We want to stress that *CrasyDSE* can not be considered a black box. In order to successfully use it, the user has to understand many of the employed routines and adapt them if required. *CrasyDSE* is merely a framework for solving DSEs that provides structures to handle Green functions and their DSEs and modules to perform the most basic steps like integration. However, as every problem has its own intricacies the user still has to implement many specific functions, e. g., extrapolation functions for dressing functions.

³Recently a similar program to *CrasyDSE* has become available for functional renormalization group equations with the program *FlowPy* [22].

3. General formulation of the numerical problem

As already mentioned DSEs form an infinitely large set of equations. For numeric calculations we take a subset of these equations, but they will always depend on Green functions whose DSEs are not part of the truncation. Depending on the details of our truncation scheme we can either provide expressions for them as external input or drop diagrams containing such Green functions. Furthermore, every Green function can consist of several dressing functions and before we can do any calculation we have to project the DSEs such that we deal with scalar integrals. Sometimes it is not possible or not feasible to project directly onto the dressing functions and additionally a linear system of equations has to be solved to get results for them. But let us for now assume for simplicity that it is possible to project directly onto the dressing functions.

Every Green function depends on several momenta. The dressing functions, however, depend only on a reduced number of variables. For example, a two-point function depends on two external momenta. Momentum conservation reduces these to one momentum. The dressing function(s) of such a Green function depend only on the remaining momentum squared, i. e., one variable instead of four. For a three-point function there are two independent momenta and the dressing functions depend on three variables. In a slight abuse of language we call the variables of the dressing functions external momenta. The number of variables is denoted as the dimension of external momenta. Later we will also encounter internal momenta. These are the remaining loop momentum variables after trivial integrations have been performed.

In the following dressing functions are denoted by $A^{g,i}(x_g)$, where g denotes the Green function to which it belongs and i labels the dressing functions of a Green function. $x_g \in \Omega_g$ represents the external momenta. We have to solve the following integral equations:

$$A^{g,i}(x_g) = A_{\text{bare}}^{g,i} + \sum_l Z^{g,i,l} \int_{\mathbb{R}^{d_l}} d^{d_l} y F_l(y, x_g, \{A\}, \{A_{\text{model}}\}) , \quad (1)$$

$$A^{g,i} : \quad \Omega_g \subseteq \mathbb{R}^{d_g} \rightarrow \mathbb{R} ,$$

where $A_{\text{bare}}^{g,i}$ are the bare dressing functions (if non-zero and possibly including a renormalization constant). The sum over l denotes contributions from different graphs and $Z^{g,i,l}$ are the renormalization constants of the bare n -point function of a given graph. The integration is over the loop momenta y . The dimensions of external and internal momenta are indicated by d_g and d_l , respectively. The $F_l(y, x_g, \{A\}, \{A_{\text{model}}\})$ denote the kernels of the integrals. They depend on the internal and external momenta explicitly and via several Green functions also implicitly. Some of them, the $\{A\}$, are a dynamic part of the truncation, whereas others, the A_{model} , are given by external input.

Before solving this system of equations numerically the following steps are required:

1. The dressing functions $A^{g,i}$ have to be approximated, e. g., by discretization of the argument or expansion in an orthogonal set of functions (see section 4.1).
2. If the integrals are divergent, one needs a regularization prescription, e. g., a sharp cutoff in $|y|$ or BPHZ [23–25] (see section 5.1).

3. Expressions for the dressings A_{model} need to be provided (see sections 4.3 and 5.1).
4. A renormalization procedure needs to be defined to fix the renormalization constants $Z^{g,i,l}$ (see sections 4.3 and 5.1 and Appendix B).

After this is settled, the integrals can be evaluated with the provided quadratures (see section 4.2). A solution can be found, for example, by fixed point iteration or Newton’s method, see, e. g., [26–28]. Both these solution methods are implemented in *CrasyDSE*, see section 4.3.

4. Implementation in C++

Solutions to the different problems stated in section 3 are implemented in *C++* in three modules:

- *dressing.cpp/hpp*,
- *quadrature.cpp/hpp*,
- *DSE.cpp/hpp*,

where *dressing.cpp/hpp* uses the structure `dse` from *DSE.cpp/hpp*.

Additionally some simple general functions are stored in *function.cpp/hpp*. All these files can be found in the directory `main`. The provided examples are located in separate directories in `examples`. They are initialized and called in the files *sphere_main.cpp*, *interp_main.cpp*, *scalar_main.cpp* and *YM4d_main.cpp*. A very basic *Makefile* for *Unix* using the *g++* GNU compiler is provided with each of the examples.

The provided modules are as self-contained as possible and can be arbitrarily extended, e. g., by including further interpolations in *dressing.cpp/hpp*, adaptive integration algorithms in *quadrature.cpp/hpp* or additional solving strategies in *DSE.cpp/hpp*.

4.1. Approximation (*dressing.cpp/hpp*)

All functions and parameters relevant for approximating the dressing functions are referenced to or stored in the structure `struct dse` defined in *DSE.hpp*. A summary together with the corresponding objects in a DSE is given in table C.6.

The expansion coefficients for the `int dim_A` dressing functions are contained in the array `double *A`. We have `int dim_x` external variables, and the numbers of expansion coefficients for each of them are saved in the array `int *n_A`. In the case of linear interpolation the interpolation points have to be stored in the array `double *x_A`. The total number of expansion coefficients is `int ntot_A := \prod_{i=0}^{dim_x-1} n_A[i]`. Since future extensions for non-zero temperature calculations require also discrete variables, i. e., Matsubara frequencies, part of the variables can be considered as integer numbers z of dimension `int dim_mat`. Currently non-zero temperature is not fully implemented and thus `dim_mat` should always be set to zero. The allocation/deallocation of the arrays `A` and `x_A` is done with `void init_A_xA(void`

`*dse_param)/void dealloc_A_xA(void *dse_param)`⁴. A minimal `dse` structure can be initialized with `void init_default_dse(void *dse_param)`. More detailed information on `A` and `x_A` can be found in Appendix A.

In `dressing.cpp/hpp` two ways to express the dressing functions are provided:

- linear interpolation (`Lin_gen_dress_interp`),
- expansion in Chebyshev polynomials (`Cheb_gen_dress_interp`).

In the case of a linear interpolation `A` contains the function values on a rectilinear grid stored (together with the discrete arguments) in `x_A` whereas in the case of an expansion in Chebyshev polynomials the expansion coefficients are stored in `A` and only the discrete arguments need to be stored in `x_A`⁵. For a Chebyshev expansion, as introduced for dressing functions of Green functions in [26], it is possible to either transform the standard interval $[-1, 1]$ linearly or logarithmically to the actual domain of interpolation via setting `int cheb_trafo[i]` to 1 or 2, respectively, where `i` denotes the external variable. By default it is set to 1 in `init_A_xA`. Furthermore, with `int cheb_func_trafo[i]` one can expand the logarithm of a function [26] instead of the function itself by changing its default value 0, set in `init_A_xA`, to 1. `i` denotes here the dressing function. For an example see section 6.2.

The provided interpolation algorithms work only as long as the arguments x of the dressing function are inside the user-defined domain Ω_g . To determine if $x \in \Omega_g$ the user-defined function `void def_domain(double *x, void *dse_param)` is called. For details of how to construct this function see Appendix B. If $x \notin \Omega_g$, the user has to provide a function for extrapolation via `double interp_offdomain(int *pos, double *x, int iA, void *dse_param)`. Details can again be found in Appendix B, but the gist is that the array `pos` knows on which side of the allowed interval $[a_i, b_i]$ the external variable x_i lies: If $x_i < a_i$ or $x_i > b_i$, `pos[i]` is 1 or 2, respectively.

The correct initialization and definition of the required parameters and functions is illustrated by the example `interp_main.cpp` interpolating three functions linearly and with Chebyshev polynomials. These functions have two discrete and three continuous variables where the domains of the continuous variables depend on the values of the discrete variables.

4.2. Integration (`quadrature.cpp/hpp`)

All functions and parameters relevant for integration are referenced to or stored in the structure `struct quad`. It is defined in `quadrature.hpp` and allocation and deallocation is done with `void init_quad(void *quad_param)` and `void dealloc_quad(void *quad_param)`, respectively. An overview of the members of `quad` relevant to the user and their usage in the context of DSEs is given in table C.7.

This module provides the means to integrate `nint` integrals of dimension `dim`. Any integral is split into three parts: a constant factor independent of any variables, a Jacobian and the remaining integrand. These three parts have to be given

⁴In order not to overload the text we refrain in most cases from a detailed explanation of all arguments and only give them for reference. Details on the meaning of the arguments can be found directly in the code where all of them are explained in the function descriptions.

⁵Note that when using the DSE solving routines also for Chebyshev interpolation the continuous external momenta have to be stored in `x_A` using the function `Cheb_init_cont_xA` described in Appendix A.1.

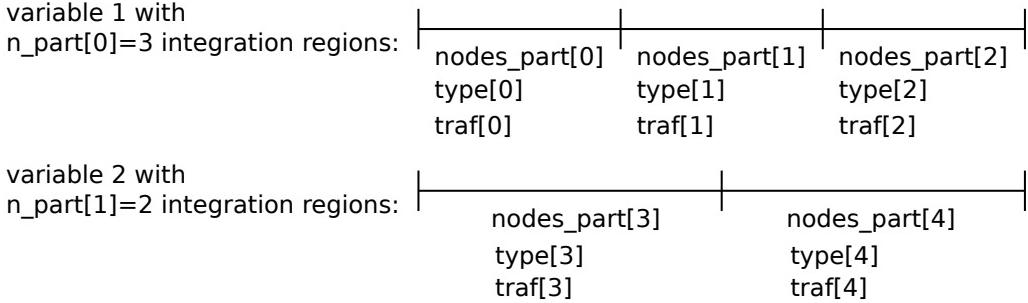


Figure 2: Example of how integration regions and related variables are used: For two integration variables the array `n_part` contains the information how many integration regions there are for each variable. The resulting integration regions are then numbered consecutively. The number of integration points, the integration type and the transformation type of each region are stored in the arrays `nodes_part`, `type` and `traf`.

by the three functions `void coeff(double *coefficients, void *int_param)`, `double jacob(double *x)` and `void integrand(double *erg, double *x, void *int_param)`. They have to be defined by the user, but `integrand` and `coeff` are usually created with the *CrasyDSE Mathematica* notebook. Further details on these functions can be found in Appendix B.

The boundaries of the integrals are defined in the function `void boundary(double *bound, double *x, int idim, void *int_param)`. Details on its required contents are provided in Appendix B. For now it suffices to say that it defines the domains $[a_i, b_i]$ of the integration variables y_i , where the inner integration boundaries may depend on the outer integration variables:

$$\int_{a_0}^{b_0} dy_0 \int_{a_1(y_0)}^{b_1(y_0)} dy_1 \cdots \int_{a_{\dim-1}(y_0, \dots, y_{\dim-2})}^{b_{\dim-1}(y_0, \dots, y_{\dim-2})} dy_{\dim-1}. \quad (2)$$

Boundaries can also depend on the external momenta. However, in this case the integration routine becomes slower. If this is required one has to set `int bound_type` to 1. The default value, set in `init_quad`, is 0, i. e., the boundaries must not depend on the external momenta. As an example where the boundaries depend on the external momenta one can consider the integration of the Yang-Mills system in section 6.

The integration of every variable y_i can be split into several parts which may increase the precision of the results, see, e. g., [26]. The number of integration regions for the i -th integration variable is given by `int n_part[i]`. Each region is labeled here by j and the number of its integration points is set by `int nodes_part[j]`. Fig. 2 illustrates the numbering of integration regions. For each region a quadrature rule has to be chosen via setting `int type[j]` to a number corresponding to one of the quadrature rules given in table 1, where also necessary parameters are indicated. The quadrature rules are defined on the interval $[-1, 1]$, which can be transformed with various functions to the actual integration interval defined in `boundary`. This works by setting `int traf[j]` to one of the values indicated in table 2.

Finally we want to draw attention to the fact that the integration is the most costly part of solving DSEs. Therefore a parallelization of the program is most

quadrature rule	<code>type[j]</code>	parameters
Gauss-Legendre	0	none
Gauss-Chebyshev type two	1	none
Fejers second rule	2	none
Double exponential	3	<code>int param[0]: stepsize</code>

Table 1: Currently implemented quadrature rules, the corresponding values of `type[j]` and the required parameters. Details can be found in `quadrature.cpp`.

transformation rule	<code>traf[j]</code>
none	0
linear	1
logarithmic	2
modified logarithmic	3
modified logarithmic [29]	4

Table 2: Currently implemented transformation rules from $[-1, 1]$ to the actual integration interval and the corresponding values for `traf[j]`. Details on these transformations can be found in the function `nw_trafo` of `quadrature.cpp`.

efficient in the function `void integrate(double *erg, void *quad_param, void *int_param)` of the quadrature module, where the loop over the internal or external momenta can be distributed to several cores. This is not implemented but a user familiar with parallelization should be able to extend the program in this direction without too much effort.

To summarize the user has to provide the following functions:

- **integrand**: Defines the integrand of the integral. Usually this will be the kernel function created by the *Mathematica* notebook.
- **coeff**: Constant factor. Usually it is created by the *Mathematica* notebook.
- **jacob**: Defines the Jacobian of the integral measure.
- **boundary**: Initializes the boundaries $a_0, b_0, \dots, a_{dim-1}(y_0, \dots, y_{dim-2})$ and $b_{dim-1}(y_0, \dots, y_{dim-2})$, where each boundary can depend on previous integration variables.

The correct initialization and definition of the needed parameters and functions is illustrated by a simple example. In `sphere_main.cpp` the function

$$\begin{aligned} \text{sphere_integrand} : \mathbb{R}^3 &\rightarrow \mathbb{R}^3 , \\ (x, y, z) &\mapsto \left(1, x^2 + y^2, (x^2 + y^2)^2\right) , \end{aligned} \tag{3}$$

times the Jacobian r is integrated over

$$\int_{-R}^R dz \int_{-\pi}^{2\pi} d\phi \int_0^{\sqrt{R^2 - z^2}} dr. \tag{4}$$

Furthermore, we exemplify here the use of external parameters. For solving DSEs the external parameters are normally the external momenta. However, the integration routine can handle also other cases of external parameters. In general one can define `int nint_para` different parameters initialized in `void init_para(int i, void *int_param)` for which the integral is performed by calling the integration routine once. For the standard application in DSEs these functions are set automatically as required by the function `init_A_xA`. In the example `sphere_main.cpp` another possibility is demonstrated. Here `init_para` is set to `sphere_init_para` which sets the external parameters as multiplicative factors for the integrals.

4.3. Solving DSEs (`DSE.cpp/hpp`)

Assuming that together with the quadrature a proper regularization has been chosen all the integrals in Eq. (1) are known and we are left with the task of solving the given integral equations including a proper renormalization.

Every Green function present in our truncated set of DSEs is represented by its own structure `dse` which contains all the necessary functions and parameters in order to evaluate its dressing functions via the function `dress`. This is already all one needs to initialize the modeled Green functions, whereas those we are going to solve for need additional information in their structures. A summary of all variables and functions relevant for the user is given in table C.8. The fact that the equations are coupled and the iteration of one dressing function needs information about dressing functions of `int n_otherGF` other Green functions is handled by `struct dse *otherGF` which contains copies of the needed structures, called when evaluating the integration kernels. Therefore it is necessary to allocate the arrays in all other Green functions before copying them to `otherGF` such that the correct pointer addresses are available. Only variables which are not supposed to be changed during the iteration procedure will be copied by value. Additionally some model parameters might be needed by the (modeled) dressing functions which are pointed to by every structure `dse` via `struct mod`, defined by the user. For an example see Fig. 3, where also the `quad` structures are indicated which contain specifics on the integration routines needed to evaluate the loop integrals in the graphs.

Focusing now on one specific Green function these graphs are grouped into `int n_looporder` contributions where all `int n_loop[i]` members of one of the groups can be evaluated using the same quadrature `struct quad Q[i]`. The evaluation of the self-energy for the `int dim_A` dressing functions and `int ntot_A` different array points x_g is then performed by calling `void selfenergy(void *dse_param)`. The result is stored in `double *self_A`. It will be used in the user-defined function `void renorm(void *DSE)` to obtain the new dressing functions, see Appendix B for details.

4.3.1. Iteration

We implemented a fixed point iteration solution technique, i. e., the system is solved by calculating

$$A_{(k+1)}^{g,i}(x_g) = A_{(k),\text{bare}}^{g,i} + \sum_l Z_{(k)}^{g,i,l} \int_{\mathbb{R}^{d_l}} d^{d_l} y F_l(y, x_g, \{A_{(k)}\}, \{A_{\text{model}}\}) , \quad (5)$$

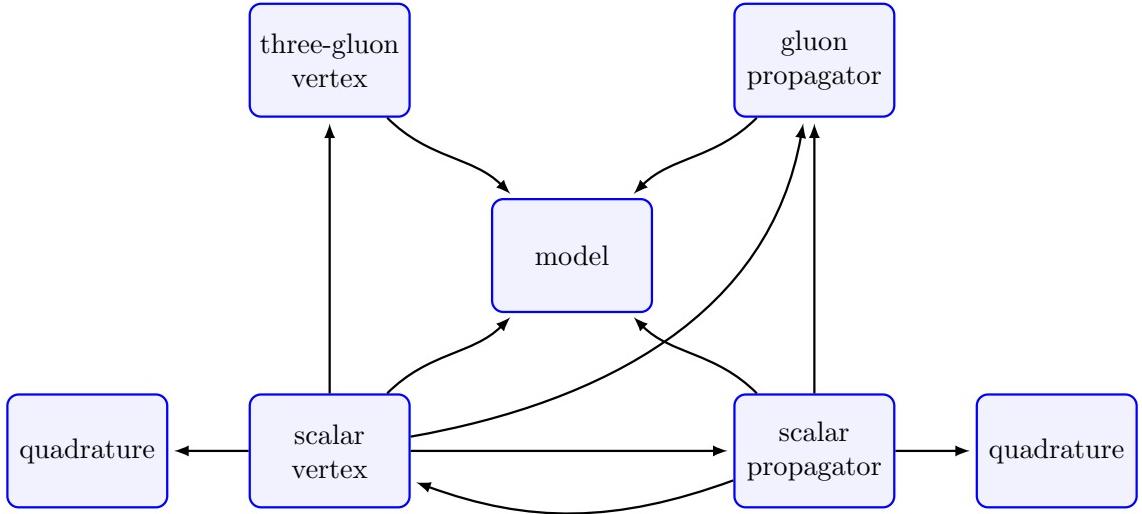


Figure 3: Schematic illustration of structures defined in the code and their dependencies for the example of Yang-Mills theory coupled to a scalar field considered in section 5.

from the previous dressing functions $A_{(k)}^{g,i}$. The calculation of the new dressing functions $A_{(k+1)}^{g,i}$ is performed in several steps, where it might be necessary that a subset of Green functions is iterated till convergence for every ‘‘meta-iteration’’ of the full set of equations.

Before starting the iterations the initial dressings $A_{(0)}^{g,i}$ of every Green function have to be set via `void init_dress(void *dse_param)`. The last step in every iteration is then to renormalize such that the $A_{(k+1)}^{g,i}$ have the correct values/derivatives at the renormalization scale(s) μ via `int renorm_n_param` parameters which may need some initialization in `void init_renormparam(void *dse_param)`. In the example of section 5 renormalization will be done by fixing the `int renorm_n_Z` renormalization constants $Z_{(k)}^{g,i,l}$ accordingly in the function `void renorm(void *DSE)`. Note that renormalization constants also appear in the bare Green functions $A_{(k),\text{bare}}^{g,i}$, but one could employ subtracted equations to drop them.

The iteration of a single Green function is performed in `void solve_iter(void *dse_param, int output)` where different stopping criteria (absolute difference `double epsabs`, relative difference `double epsrel` and maximum number of iterations `int maxiter`) are available. Several Green functions can be united in an array which can be passed to `void meta_solve_iter(struct dse *DSE, int ndse, double epsabsstop, double epsrelstop, int iterstop, int output)`. It has the same stopping criteria as `solve_iter` and in every meta-iteration `solve_iter` is called for every Green function. This allows different relative iterations, e. g., all Green functions are iterated once for every meta-iteration or a subset of Green functions is iterated till convergence while another subset is iterated only once.

To get an idea of the efficiency of our code we compared the calculation from section 5 with an independently created code that was optimized for this problem [30]. In general the difference in time depends on how much optimization is possible in the given problem. In the present case the time difference for one iteration was less than a factor of 2.

4.3.2. Newton's method

Another method for solving DSEs is based on Newton's method to solve a nonlinear system of equations. It was already used in many DSE calculations, see, for instance, [26–28, 31]. For this method the system of DSEs is rewritten into the following form:

$$E^{(i,k)} = -A^i(x_k) + A_{\text{bare}}^i(x_k) + \sum_l Z^{i,l} \int_{\mathbb{R}^{d_l}} d^{d_l}y F_l(y, x_k, \{A\}, \{A_{\text{model}}\}), \quad (6)$$

where i labels the dressing functions of all DSEs and x_k denotes the external momenta. We assume here that the dressing functions $A^i(x_k)$ are expanded in a set of basis functions. The corresponding expansion coefficients are the unknown variables $c^{(i,j)}$, where j labels the polynomials. The goal is to find those values for $c^{(i,j)}$ that make all $E^{(i,k)}$ vanish. Newton's method yields new coefficients by the following formula:

$$c'^{(i,j)} = c^{(i,j)} - \lambda \sum_{i',k} \left(J_{(i',k)}^{(i,j)} \right)^{-1} E^{(i',k)}, \quad (7)$$

where the Jacobian J is given by

$$J_{(i',k)}^{(i,j)} := \frac{\partial E^{(i',k)}}{\partial c^{(i,j)}}. \quad (8)$$

The backtracking parameter λ can be used to optimize this step by choosing an appropriate value between zero and one. The determination of λ can be subject of sophisticated algorithms, see, for example, [28]. Here, however, we simply cut λ in half if the norm of the new $E^{(i,k)}$ is not smaller than that of the old one. If the starting functions are well chosen this is sufficient for the example of section 6. This procedure is repeated until the norm of the vector $E^{(i,k)}$ drops below a given value or a maximal number of iterations is reached. A single iteration step takes here much longer than for the fixed point iteration described in section 4.3.1 because the calculation of the Jacobian is rather expensive. In principle the derivatives required to get J can be done directly, but here Broyden's method is used which defines an approximate Jacobian by a simple forward differentiation with small step size h :

$$J_{(i',k),\text{approx}}^{(i,j)} := \frac{E^{(i',k)}(c^{(i,j)} + h) - E^{(i',k)}(c^{(i,j)})}{h}, \quad (9)$$

where the notation $E^{(i',k)}(c^{(i,j)} + h)$ means that $E^{(i',k)}$ is calculated with the coefficient $c^{(i,j)}$ changed by h , whereas $E^{(i',k)}(c^{(i,j)})$ refers to the original $E^{(i',k)}$. This prescription proved very reliable for the example treated in section 6.

Newton's method is implemented in the function `void solve_iter_secant(struct dse *DSE, int ndse, int maxiter, double eps_E, int output)`. As arguments it takes the array of `dse`s `DSE`, the length of this array `ndse`, the maximal number of iterations `maxiter`, the stopping value for $\sum_{i,k} E^{(i,k)}$ `eps_E` and an integer number `output` which determines if intermediary output should be printed to the screen (1) or not (0). Of course Newton's method can be combined with the fixed point iteration. For example, one could solve two of three DSEs with Newton's method

and then iterate the third one with `solve_iter`.

5. Solving the gap and vertex equations of Yang-Mills theory coupled to a scalar field

In this section we describe how to solve a truncated set of DSEs of Yang-Mills theory coupled to a scalar field. We will first give a short overview of the employed truncation and then explain how to solve it with *CrasyDSE*.

The derivation of the DSEs is not discussed here, but we provide details in the *Mathematica* notebook *DoFun_YM+Scalar.nb*. The results of this notebook form the basis on which we create the functions for the *C++* code. The corresponding steps are contained in a second notebook, *CrasyDSE_YM+scalar.nb*. It describes how the expressions of the DSEs have to be modified so they can be used as input for *CrasyDSE*. In a second part all required definitions are provided and the kernels are created. We will explain only the latter here, since the first steps consist only of standard *Mathematica* transformations and are not special to *CrasyDSE*. Finally we explain some details for this specific example in the *C++* code. The provided files allow the interested reader to follow the complete procedure, from the derivation of the DSEs to their numeric solution, in detail.

5.1. Yang-Mills theory coupled to a scalar field

In nature elementary matter fields are fermions. In quantum chromodynamics, for example, these are the quarks which interact via gluons. However, since their spin is 1/2, quarks are Dirac fields and consequently represented by spinors. An advantage of functional methods is that they do not suffer from fundamental problems when dealing with anti-commuting fields. However, calculations are complicated by the Dirac structure, since it allows more dressing functions than for a simple scalar, see, e. g., [32]. While at the level of propagators this is still doable, see, e. g., [33–36], three-point functions become already quite tedious [37]. Since some non-perturbative phenomena like confinement may not depend on the fields being spinors or scalars, one can alleviate calculations by replacing the quarks by scalar fields. In order to mimic quarks such fields also have to be in the fundamental representation. The calculation used as an example for the presentation of *CrasyDSE* in this section is motivated by investigations along these lines [30, 38–42].

The renormalized action of this theory in Landau gauge reads in momentum space

$$\begin{aligned}
S[A, \bar{\varphi}, \varphi] = & \int \frac{d^d q}{(2\pi)^d} \left(\frac{1}{2} Z_3 A_\mu^a(q) (q^2 g_{\mu\nu} - q_\mu q_\nu) A_\nu^a(-q) \right. \\
& + \hat{Z}_3 \bar{\varphi}^i (q^2 + Z_m m^2) \varphi^i \Big) \\
& + \int \frac{d^d q_1 d^d q_2}{(2\pi)^{2d}} \left(i Z_1 g f^{abc} q_{1\mu} A_\nu^a(q_1) A_\mu^b(q_2) A_\nu^c(-q_1 - q_2) \right. \\
& \left. + \hat{Z}_1 g T_{ij}^a (2q_{2\mu} + q_{1\mu}) A_\mu^a(q_1) \bar{\varphi}^i(q_2) \varphi^j(-q_1 - q_2) \right) + \dots , \quad (10)
\end{aligned}$$

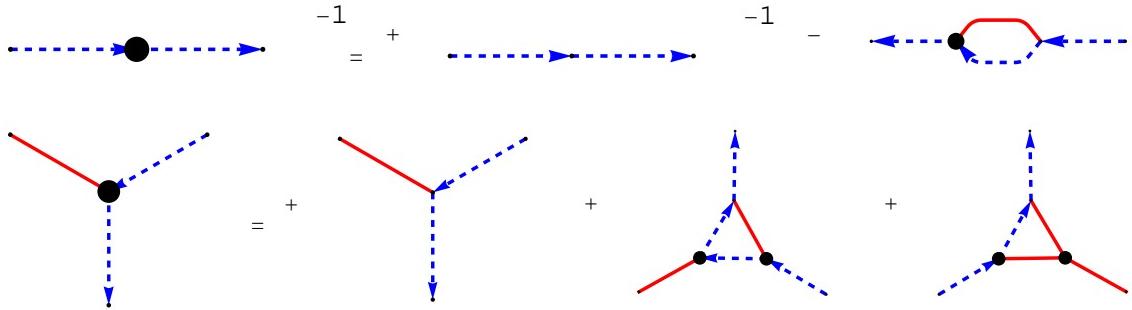


Figure 4: *Top:* The truncated DSE of the scalar two-point function. The full DSE can be found, for example, in ref. [38]. *Bottom:* The truncated DSE of the scalar-gauge field vertex. The second and third diagrams of the vertex DSE are called Abelian and non-Abelian diagrams, respectively. Gauge fields have red, continuous lines and scalar fields blue, dashed lines. Thick blobs denote dressed vertices. All internal lines are dressed propagators.

where T_{ij}^a are the Hermitian generators in the fundamental representation of $SU(N_c)$ with the structure constants f^{abc} . The dots correspond to four-point vertices and terms with Faddeev-Popov ghosts. The former are dropped in our truncation and the latter do not appear in the DSEs considered here, which are those of the scalar two-point function and of the scalar-gauge field vertex. The full DSE of the two-point function can be found, for example, in ref. [38]. Here we neglect for both DSEs all diagrams containing four-point functions which renders the scalar gap equation diagrammatically equal to the quark gap equation. This truncation is also motivated by the fact that two-loop diagrams are subleading in the UV.

In the following we will focus on the scalar sector of the theory, viz. the scalar propagator and the scalar-gauge field vertex. As can be seen from the truncated DSEs of the scalar two-point function and the scalar-gauge field vertex in Fig. 4 we have the following four quantities left in our truncation: the propagators of the scalar and the gauge fields, the three-gauge field vertex and the scalar-gauge field vertex.

The scalar propagator and the scalar-gauge field vertex are parametrized as

$$(D_s)_{mn}(p) = \frac{A_s(p^2)}{p^2} \delta_{mn}, \quad (11)$$

$$\Gamma_{mn,\mu}^{\bar{s}sA,a}(p_1, p_2) = g T_{mn}^a (A_{p_1}(p_1^2, p_2^2, z) p_{1\mu} + A_{p_2}(p_1^2, p_2^2, z) p_{2\mu}),$$

where in the case of the vertex p_1 and p_2 are the momenta of the scalar particles and $z = p_1 \cdot p_2 / (|p_1||p_2|)$. Introducing a sharp momentum cutoff Λ to regularize the self-energy contributions we need to approximate the three dressing functions

$$A_s : [0, \Lambda^2] \rightarrow \mathbb{R}, \quad (12)$$

$$A_{p_1, p_2} : [0, (\Lambda/2)^2]^2 \times [-1, 1] \rightarrow \mathbb{R}, \quad (13)$$

which will be done by linear interpolation for A_{p_1, p_2} and linear interpolation as well as Chebyshev expansion in the case of A_s . Choosing the cutoff in the vertex to be smaller by a factor of two has the effect that only the dressing functions A_{p_1, p_2}

will be called at large momenta outside their domain when evaluating the self-energy integrals. We will approximate them with their bare value $A_{p_1,p_2} = \hat{Z}_1$ where necessary.

Additional information is required for the gauge field propagator and the three-gauge field vertex. For the latter we use for simplicity

$$\Gamma_{\mu\nu\rho}^{AAA,abc}(p_1, p_2, p_3) = \frac{\tilde{Z}_3}{\tilde{Z}_3} \Gamma_{\mu\nu\rho}^{AAA,abc,(0)}(p_1, p_2, p_3), \quad (14)$$

where finiteness of the ghost-gauge field vertex in Landau gauge $\tilde{Z}_1 = 1$ and the Slavnov-Taylor identity $Z_1/\tilde{Z}_1 = Z_3/\tilde{Z}_3$ [43] have been used. The bare vertex $\Gamma_{\mu\nu\rho}^{AAA,abc,(0)}(p_1, p_2, p_3)$ is given in Eq. (20). For the dressing function $Z(p^2)$ of the Landau gauge field propagator we employ a fit to the solution of the ghost-gluon system obtained within the DSE framework provided in ref. [11] (see also section 6):

$$\begin{aligned} Z(x) &= \left(\frac{\alpha(x)}{\alpha(\mu^2)} \right)^{-\gamma} R^2(x), \\ R(x) &= \frac{c \left(\frac{x}{\Lambda_{QCD}^2} \right)^\kappa + d \left(\frac{x}{\Lambda_{QCD}^2} \right)^{2\kappa}}{1 + c \left(\frac{x}{\Lambda_{QCD}^2} \right)^\kappa + d \left(\frac{x}{\Lambda_{QCD}^2} \right)^{2\kappa}}, \end{aligned} \quad (15)$$

where $c = 1.269$, $d = 2.105$, $\gamma = -13/22$, $\kappa = 0.5953$, $\Lambda_{QCD} = 0.714$ GeV and

$$\alpha(x) = \frac{\alpha(0)}{\ln \left[e + a_1 \left(\frac{x}{\Lambda_{QCD}^2} \right)^{a_2} + b_1 \left(\frac{x}{\Lambda_{QCD}^2} \right)^{b_2} \right]}, \quad (16)$$

with $a_1 = 1.106$, $a_2 = 2.324$, $b_1 = 0.004$, $b_2 = 3.169$ and $\alpha(0) = 8.915/N_c$, N_c being the number of colors which we take to be three. Note that the renormalization constants Z_3, \tilde{Z}_3 can be calculated from the running coupling α as described in [11].

For the present system the iteration procedure is very stable and we can start from a massless bare scalar propagator $A_s \equiv 1$ and a bare scalar-gauge field vertex. As previously mentioned the integrals are regularized via an ultraviolet cutoff. To determine the renormalization constants of the scalar propagator we fix the values $A_s(\mu^2)$ and $Z_m m^2$. Furthermore the vertex will be renormalized by enforcing the Slavnov-Taylor identity $\hat{Z}_1/\tilde{Z}_3 = \tilde{Z}_3/\tilde{Z}_3$. With this prescription the system is then multiplicatively renormalizable, i. e., the expressions $\hat{Z}_3 A_s$ and $(\hat{Z}_1)^{-1} A_{p_1,p_2}$ are independent of the renormalization point μ^2 . Results confirming this for the propagator are shown in fig. 5.

We can here also illustrate how extending truncations complicates the system of equations: A simple truncation takes into account only the scalar propagator. In this case we need the gauge field propagator and the scalar-gauge field vertex as input and we only have to calculate one integral. Its integrand is comparatively simple. Going only one step further by including also the scalar-gauge field vertex requires solving in total for three dressing functions (the vertex has two and the propagator one) by calculating five integrals, whereby the complexity of the integrands has also

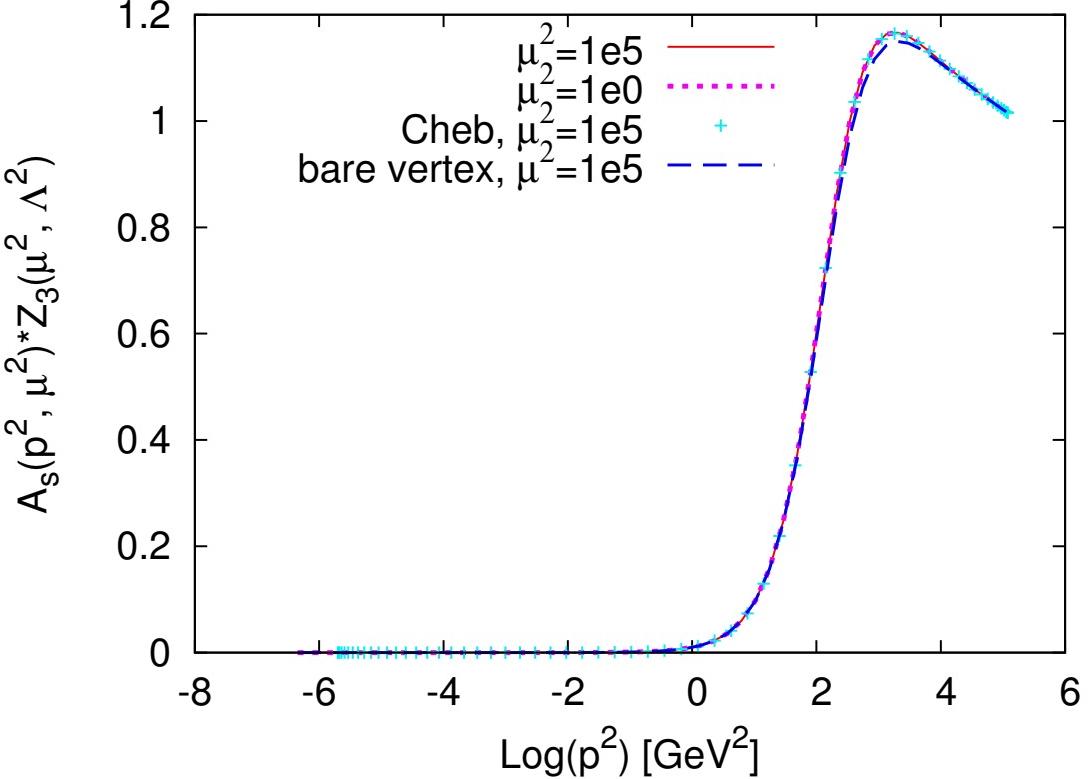


Figure 5: Dressing function of the propagator times its renormalization constant. Multiplicative renormalizability of the system propagator and vertex is clearly visible as the results obtained with two different choices of μ^2 are indistinguishable. Independence from the approximation method is seen from the results obtained with the Chebyshev expansion method. For comparison the result with a bare scalar-gauge field vertex is included.

increased considerably.

We should mention that the employed truncation is not state of the art but it is sufficient to illustrate many features of *CrasyDSE*. More elaborate results for Yang-Mills theory coupled to a fundamental scalar will be presented elsewhere [30].

5.2. Generating the C++ files with the integrands

We turn now to the creation of the kernels with *Mathematica*. The subsequent explanations follow the notebook *CrasyDSE_YM+scalar.nb*. To initialize the required functions and the expressions for the DSEs we evaluate the initialization cells with

```
FrontEndTokenExecute["EvaluateInitialization"]
```

or via the menu entry *Evaluation* → *Evaluate Initialization cells*. Now the variables containing the expressions of the integrals are defined: `gapAlgProjLoopIntegrand` is the integral of the gap equation and `vertexAbelianProjp1Final`, `vertexNonAbelianProjp1Final`, `vertexAbelianProjp2Final` and `vertexNonAbelianProjp2Final` correspond to the integrals of the Abelian and non-Abelian di-

ograms projected onto the external momenta p_1 and p_2 . Furthermore the package *CrasyDSE* is loaded.

Basically for the generation of the *C++* files only one function is needed. However, before we can use it we need to define several expressions. They are split into lists containing parameters, momentum variables and dressing function names. In the following the order of the elements in lists is very important. Thus one has to be very careful when one changes something later on, because this could lead to inconsistencies with the *C++* code.

In the gap equation two parameters appear: the number of colors N_c and the coupling constant g . We put them both into a list:

```
parasGap = {Nc, g};
```

Furthermore we specify the external ($\text{ps} := p^2$) and internal ($\text{qs} := q^2$, $\text{ct} := \cos(\varphi) = p \cdot q / |p||q|$) variable names:

```
extVarsGap = {ps};
intVarsGap = {qs, ct};
```

Although we did not introduce abbreviations of momentum combinations for the gap equation, we need to define a variable for this, because we will need it later:

```
extraVarsCListGap = {};
```

Finally we have to specify which dressings appear in each equation. They are separated into two lists, one for the dressings of the Green function we are calculating and one for all other dressings belonging to other Green functions. The propagator has only one dressing function D_s , so the first list contains one item only:

```
dressingsGap = {Ds};
```

But the gap equation also depends on other dressing functions, namely on the one of the gauge field, D_A , and on the two of the scalar-gauge field vertex, $D_{A\bar{s}}^{(1)}$ and $D_{\bar{A}s}^{(1)}$. All dressings belonging to the same Green function have to be grouped together in one sublist:

```
otherGreenFuncsGap = {{DA}, {DAssb1, DAssb2}};
```

These are the lists required for the gap equation as input for generating the *C++* files.

The lists for the vertex equation have the same structure and we only list them here:

```
parasVertex = {Nc, g};
extVarsVertex = {p1s, p2s, ca};
intVarsVertex = {qs, ct1, ct2};
extraVarsCListVertex = {{p1p2, ca Sqrt[p1s p2s]}, 
    {p1q, Sqrt[p1s qs] Cos[ct2]}, 
    {p2q, (ca ct2 + Sqrt[1 - ca^2] ct1 Sqrt[1 - ct2^2]) Sqrt[p2s qs]}, 
    {p1mq, p1s - 2 p1q + qs}, 
    {p2mq, p2s - 2 p2q + qs}, 
    {p1mp2s, p1s + p2s - 2 Sqrt[p1s p2s] ca}};
dressingsVertex = {DAssb1, DAssb2};
otherGreenFuncsVertex = {{DA}, {Ds}, {DAAA}};
```

Note that the vertex has two dressings by itself and depends on three other Green functions. Furthermore we provided with the list `extraVarsCListVertex` the definitions of employed abbreviations, e. g., $p1p2 := p_1 \cdot p_2 = \cos(\alpha)|p_1||p_2|$.

Before we generate the *C++* files we split off numeric coefficients of the integrands. We call the resulting expressions *kernels* and *coefficients*. Instead of doing this by hand, we use the function `splitIntegrand`. It takes as arguments an expression and a list of variables. Everything in the overall factor that does not contain a variable will be put into the coefficient:

```
{coeffGap, kernelGap} =
  splitIntegrand[gapAlgProjLoopIntegrand,
    Join[extVarsGap, extraVarsCListGap[[All, 1]], intVarsGap]] /.
  Z1h :> 1 // Simplify

--> {(g^2 (-1 + Nc^2))/(8 Nc \[Pi]^3), (1/pplusqs)(1 - ct^2)^(3/2)
  DA[qs] (DAssb1[ps,
    pplusqs, -((ps + ct Sqrt[ps qs])/Sqrt[pplusqs ps])] -
  DAssb2[ps,
    pplusqs, -((ps + ct Sqrt[ps qs])/Sqrt[pplusqs ps])]) Ds[pplusqs]}
```

We discarded the renormalization function `Z1h` here since its implementation is handled manually.

For the vertex we do the same but bear in mind the following structure: Both for coefficients and kernels every loop integral is treated as a single expression, and for every equation all loops are grouped into lists. The syntax of the list of coefficients or kernels is thus

```
{loop 1 of eq. 1, loop 2 of eq. 1, ...},
{loop 1 of eq. 2, loop 2 of eq. 2, ...}, ...}
```

For the first and second projections we split the integrands as follows:

```
{coeffsVertexProjp1, kernelsVertexProjp1} = Transpose[
  splitIntegrand[#, Join[extVarsVertex, extraVarsCListVertex[[All, 1]],
    intVarsVertex]] & /@ {vertexAbelianProjp1Final,
  vertexNonAbelianProjp1Final} /. {Z1h :> 1, Z1 :> 1} // Simplify];
{coeffsVertexProjp2, kernelsVertexProjp2} =
  Transpose[
  splitIntegrand[#, Join[extVarsVertex, extraVarsCListVertex[[All, 1]],
    intVarsVertex]] & /@ {vertexAbelianProjp2Final,
  vertexNonAbelianProjp2Final} /. {Z1h :> 1, Z1 :> 1} // Simplify];
```

Again we have discarded the renormalization functions. The final lists of kernels and coefficients are

```
kernelsVertex = {kernelsVertexProjp1, kernelsVertexProjp2};
coeffsVertex = {coeffsVertexProjp1, coeffsVertexProjp2}
```

```
--> {{g/(16 Nc Pi^3), -((g Nc)/(32 Pi^3))},
  {g/(16 Nc Pi^3), -((g Nc)/(32 Pi^3))}}
```

We show the coefficients explicitly. One can easily spot the $1/N_c$ and N_c dependences of the Abelian and non-Abelian diagrams, respectively.

Finally we have everything to generate the *C++* code. We do so with the function `exportKernels`:

```
exportKernels[{FileNameJoin[{NotebookDirectory[], ".."}],  
  "kernelsAll"},  
 {"scalar_QCD.hpp"},  
 {{"coeffGap", "kernelGap"},  
  {coeffGap},  
  {kernelGap},  
  dressingsGap,  
  otherGreenFuncsGap,  
  parasGap,  
  extVarsGap,  
  intVarsGap,  
  extraVarsCListGap},  
 {{"coeffsVertex", "kernelsVertex"},  
  coeffsVertex,  
  kernelsVertex,  
  dressingsVertex,  
  otherGreenFuncsVertex,  
  parasVertex,  
  extVarsVertex,  
  intVarsVertex,  
  extraVarsCListVertex}  
 ]]
```

It will create two files *kernelsAll.hpp* and *kernelsAll.cpp*. The filenames are determined by the first argument where we also indicate that the files should be exported to the parent directory. The second argument here is the name of an additional header file which contains functions specific to this example. The third argument contains all the information we gathered above: It is a list where every item corresponds to one DSE. For every DSE we have the following entries:

- The *C++* names of the functions containing the coefficients and the kernels.
- A list with the expression(s) for the coefficient(s).
- A list with the expression(s) for the kernel(s).
- The list of dressings for this Green function.
- The list of dressings from other Green functions.
- The list of parameters.
- The list of external variables.
- The list of internal variables.
- The list of extra variables.

Note that without specifying a path in the first argument of `exportKernels` the files will be created in the directory of the notebook.

We want to mention here the function `functionToString`, which is used by `exportKernels` but can also be used directly by the user. It creates a string of the expression given as its argument similar to the *Mathematica* function `CForm`, but it replaces some common functions like `Power` or `Sin` by its *C++* counterparts `pow` or `sin`. If a function is not included, it can be added by hand, for example:

```
functionToString[a^b + Sin[b]/10 - 5 Sinh[a], {Sinh :> sinh}]

--> 0.1*(sin(b)) + -5.*(sinh(a)) + (pow(a, b))
```

This finishes our work in *Mathematica* and we proceed with the *C++* code.

5.3. Numerical code

The *C++* code, contained in `scalar_main.cpp`, is extensively commented and every variable that appears is described directly in the file. Here we only give a rough overview of the required initializations.

In the file `scalar_main.cpp` first the model and then all Green functions are initialized. For the former the definitions are as simple as providing numeric values for some parameters, e. g., $N_c = 3$. All Green functions are defined as a `dse` structure, which contains a pointer to `mod` which is reserved for hosting model parameters, see also Fig. 3. Since the gauge field propagator and the three-gauge field vertex are given by ansätze the only additional information these structures need are the corresponding definitions. The dynamically calculated Green functions also contain an array of `quad` structures, namely one for each different integration. Consequently variables like the numbers of integration points and the quadrature types have to be initialized. We also have to provide starting expressions for the dressings and information on the other Green functions contained in a DSE - handled via the array `otherGF`. For example, for the gap equation these are the gauge field propagator and the scalar-gauge field vertex. In the *C++* code dressing functions do not have a specific name, but are just collected in the function `dress` where it is important at all times to maintain the same assignment of the dressing functions as in the notebook. The integrands of the self-energy are defined in the kernels file created with `CrasyDSE_YM+scalar.nb`. Also a renormalization procedure has to be defined. Finally, the iteration is done with the function `meta_solve_iter`.

6. Landau gauge Yang-Mills theory

As a second example we use pure Yang-Mills theory which requires some different methods. The most obvious change is that we use a Newton's method instead of a direct fixed point iteration. Again we provide the complete *Mathematica* and *C++* code together with the program. However, as the creation of the kernel files is rather similar to the case of the previous section we refrain from showing any details.

6.1. Truncation and ansätze

As in the last section we will employ the Landau gauge. The DSE system truncated at the level of propagators has been investigated with DSEs for some time now, see, for example, [10, 12, 14, 27, 31, 44]. We will here reproduce the

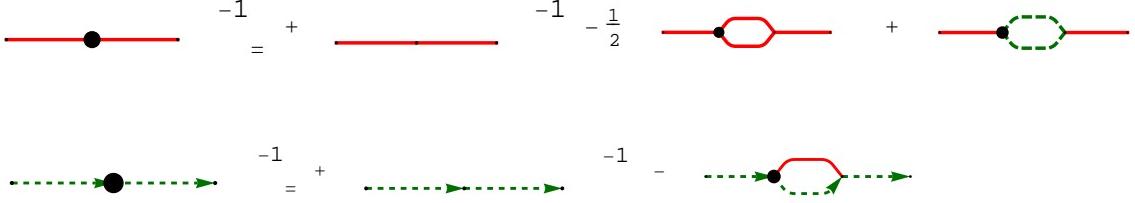


Figure 6: The truncated two-point DSEs of pure Yang-Mills theory. Gluons have red, continuous lines and ghost fields green, dashed lines. Thick blobs denote dressed vertices. All internal lines are dressed propagators.

solutions of refs. [12, 31, 45]. Besides employing a Newton procedure to solve this set of equations another difference to the previous section lies in the renormalization procedure: Here we work with subtracted DSEs.

The system we investigate consists of the ghost and gluon two-point DSEs. The former is used without change, while the gluon DSE is truncated [10, 14]: We neglect all diagrams involving a bare four-gluon vertex, i.e., the tadpole diagram and all two-loop diagrams. They are subleading in the UV and it was shown analytically for the scaling solution that they are also subleading in the IR [46]. The remaining unknown quantities are the ghost-gluon and the three-gluon vertices for which we use suitable ansätze. The truncated set of DSEs is depicted in Fig. 6.

The ghost and gluon propagators are given by

$$D_{\mu\nu}^{ab}(p) = \delta^{ab} \left(g_{\mu\nu} - \frac{p_\mu p_\nu}{p^2} \right) \frac{Z(p^2)}{p^2}, \quad (17)$$

$$G^{ab}(p) = -\delta^{ab} \frac{G(p^2)}{p^2}. \quad (18)$$

For the ghost-gluon vertex $\Gamma_\mu^{A\bar{c}c,abc}(p_1, p_2, p_3)$ the bare version,

$$\Gamma_\mu^{A\bar{c}c,abc,(0)}(p_1, p_2, p_3) = i g f^{abc} p_{2\mu}, \quad (19)$$

is used as motivated originally by an argument of Taylor. However, several studies both on the lattice [47] and in the continuum [48, 49] confirmed this to be a very reliable ansatz. For the full three-gluon vertex $\Gamma_{\mu\nu\rho}^{AAA,abc}(p_1, p_2, p_3)$ we use the tensor structure of the bare vertex $\Gamma_{\mu\nu\rho}^{AAA,abc,(0)}(p_1, p_2, p_3)$ amended by a dressing $D^{AAA}(p_1^2, p_2^2, p_3^2)$ that guarantees the correct UV behavior of the gluon dressing function [31]:

$$\Gamma_{\mu\nu\rho}^{AAA,abc,(0)}(p_1, p_2, p_3) = i g f^{abc} (g_{\mu\nu}(p_2 - p_1)_\rho + g_{\nu\rho}(p_3 - p_2)_\mu + g_{\rho\mu}(p_1 - p_3)_\nu), \quad (20)$$

$$\Gamma_{\mu\nu\rho}^{AAA,abc}(p_1, p_2, p_3) = \Gamma_{\mu\nu\rho}^{AAA,abc,(0)}(p_1, p_2, p_3) D^{AAA}(p_1^2, p_2^2, p_3^2), \quad (21)$$

$$D^{AAA}(p_1^2, p_2^2, p_3^2) = \frac{1}{Z_1} \frac{(G(p_2^2)G(p_3^2))^{1-a/\delta-2a}}{(Z(p_2^2)Z(p_3^2))^{1+a}}. \quad (22)$$

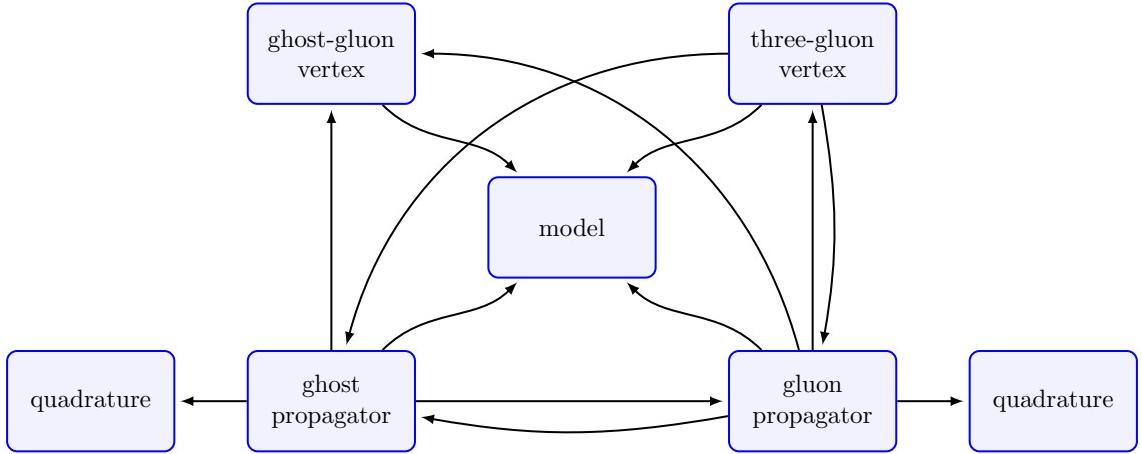


Figure 7: Schematic illustration of structures defined in the code and their dependencies for the pure Yang-Mills system.

a is a parameter chosen as 3δ , where δ is the anomalous dimension of the ghost propagator. Z_1 is the renormalization constant of the three-gluon vertex. The dependencies of all Green functions on each other are shown in Fig. 7.

An important issue of the gluon DSE are spurious quadratic divergences. They appear because we employ a numerical cutoff as UV regularization which breaks gauge invariance. There are several ways to deal with them, see, for example, [12, 31, 50]. Here we subtract an additional term in the kernel of the gluon loop in the gluon DSE [31]. The derivation of the DSEs with *DoFun* and the projection to scalar quantities are described in the notebook *DoFun_YM_4d.nb* and the creation of the kernel files in *CrasyDSE_YM_4d.nb*. For details we refer to them.

6.2. Renormalization and solution

For the present system we will use subtracted DSEs, i. e., we subtract from a DSE at external momentum p the DSE at a fixed external momentum p_0 :

$$D^{-1}(p^2) = Z^{-1} + \Pi(p^2) \quad \Rightarrow \quad D^{-1}(p^2) = D^{-1}(p_0^2) + \Pi(p^2) - \Pi(p_0^2). \quad (23)$$

Thus we can trade the renormalization constant Z for specifying the value of the dressing function at the subtraction point p_0 . For the gluon propagator we choose the subtraction point p_0 at sufficiently high momenta since we expect the two-point function to be divergent at low momenta. For the ghost, however, it is most advantageous to specify the dressing at zero momentum. These conditions are boundary conditions for the integral equations. As it turns out two different types of solutions can be found depending on the value of the ghost dressing at zero momentum: Choosing finite values a solution of the family of decoupling solutions emerges [12, 44, 51], while with an infinite zero-momentum dressing we get the scaling solution [10, 12]. The former has a finite gluon propagator and a finite ghost dressing function at zero momentum and the latter an IR vanishing gluon propagator and an IR divergent ghost dressing function. Thereby the divergence of the ghost dressing and the vanishing of the gluon dressing can be described by power laws whose exponents δ_{gh} and δ_{gl} , respectively, are related by $\delta_{gl} + 2\delta_{gh} = 0$, whereby $\delta_{gh} := \kappa = 0.595353$ can be calculated analytically [48, 52].

There are several choices at which point of the calculation the subtraction of a DSE can be performed. For illustration purposes we employ a different one for each propagator: For the ghost we use the subtracted expression in the kernel file. This is advantageous because the limit of vanishing external momentum can be done analytically but is problematic numerically. For the gluon propagator, on the other hand, we only create the unsubtracted expressions. The subtraction is then performed with a function in *C++*. Here the subtraction is numerically unproblematic.

The specific renormalization procedure has to be taken into account in the renormalization function in the *C++* code. Furthermore, it is important to note that this function does not calculate the right-hand side of a DSE as in the example of the scalar system but the difference between the right- and left-hand side of a (subtracted) DSE:

$$E(p^2) := -D^{-1}(p^2) + D^{-1}(p_0^2) + \Pi(p^2) - \Pi(p_0^2). \quad (24)$$

The reason is the employed Newton procedure as described in section 4.3.2 which attempts to bring $E(p^2)$ to zero. $E(p^2)$ is calculated for every external momentum and saved as an array of the DSE structure.

The behavior of the dressing functions in the IR and UV is known analytically. In the intermediate regime, between two given momenta ϵ and Λ , above and below the IR and UV cutoffs, respectively, they are expressed by an expansion in N Chebyshev polynomials⁶:

$$G_{IM}(p^2) = \exp \sum_{i=0}^{N-1} c_i^{(gh)} T_i(M(p^2)), \quad (25)$$

$$Z_{IM}(p^2) = \exp \sum_{i=0}^{N-1} c_i^{(gl)} T_i(M(p^2)), \quad (26)$$

where $M(p^2)$ maps the regime $[\epsilon, \Lambda]$ to $[-1, 1]$. For momenta below ϵ we employ a power law with the given exponent and the coefficient calculated from the lowest known point in the Chebyshev expansion:

$$G_{IR}(p^2) = A^{(gh)} (p^2)^{\delta_{gh}}, \quad (27)$$

$$Z_{IR}(p^2) = A^{(gl)} (p^2)^{\delta_{gl}}. \quad (28)$$

For momenta higher than Λ an extrapolation in agreement with the UV behavior is chosen:

$$G_{UV}(p^2) = G(s^2)(w \log(p^2/s^2) + 1)^\delta, \quad (29)$$

$$Z_{UV}(p^2) = Z(s^2)(w \log(p^2/s^2) + 1)^\gamma. \quad (30)$$

s is the highest momentum at which the Chebyshev expansion is known, δ or γ are the anomalous dimensions of the ghost and gluon, respectively, and $w = 11 N_c \alpha(s) G(s)^2 Z(s)/12\pi$. $\alpha(p^2)$ is a possible non-perturbative definition of the

⁶The exponential is chosen due to better convergence properties. For such an expansion see also, for example, ref. [26, 28].

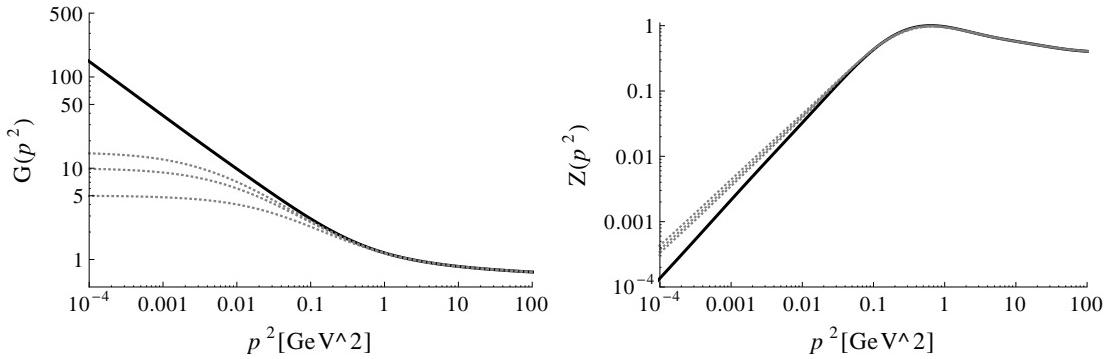


Figure 8: Ghost (left) and gluon (right) dressing functions. The continuous line corresponds to the scaling solution, the dashed lines are solutions of the decoupling type.

running coupling [53, 54]:

$$\alpha(p^2) := \alpha(\mu^2)G(p^2)^2Z(p^2). \quad (31)$$

The value of $\alpha(\mu)$ is an input parameter and sets the scale: At μ we have $G(\mu^2)^2Z(\mu^2) = 1$.

As starting functions for the propagator dressings in the intermediate regime we use

$$Z_{ans}(p^2) = f_{IR}(p^2)^2(p^2)^{\delta_{gl}} + c_{UV} \cdot f_{UV}(p^2), \quad (32)$$

$$G_{ans}(p^2) = f_{IR}(p^2)(p^2)^{\delta_{gl}} + 1, \quad (33)$$

with the IR and UV damping factors given by

$$f_{IR}(p^2) = \frac{L_{IR}}{L_{IR} + p^2}, \quad (34)$$

$$f_{UV}(p^2) = \left(\frac{p^2}{L_{UV} + p^2} \right)^2, \quad (35)$$

where L_{IR} and L_{UV} are dimensionful parameters conveniently set to 1. The parameter c_{UV} can be used to adjust the starting function in order to speed up convergence. Here it is chosen as 1. These ansätze respect the qualitative IR behavior which leads to a faster convergence than starting with constant functions. In general the Newton procedure becomes more stable when using starting functions close to the solution. The starting functions being not differentiable at the UV matching poses no problem.

For the integration we used Gauss-Legendre quadratures. Furthermore it was advantageous to split the radial integration at the value of the external momentum which requires setting `bound_type` of all quadratures to 1. Note that this has to be done after `init_quad`, which sets the default value 0. This allows a higher precision of the final result. However, this comes at the prize of slowing down the integration as the integration boundaries have to be calculated for every external momentum.

In Fig. 8 we show the results of the calculations for different boundary conditions of the ghost. It is clearly visible that all solutions coincide in the UV and only show their distinct behavior in the IR. In table 3 we provide the input parameters used

parameter	value	parameter	value
N_c	3	κ	0.595353
$\alpha(\mu^2)$	1, 0.5	h	10^{-3}
UV cutoff	10^3	ϵ	2×10^{-8}
IR cutoff	10^{-12}	Λ	0.99×10^3
δ	$-9/44$	gluon subtraction point p_0	1.2
γ	$-13/44$	value of gluon dressing at p_0	0.93
L_{IR}	1	value of ghost dressing at $p = 0$	0, 5, 10, 25
L_{UV}	1		

Table 3: Parameters for the calculation. Where several values are given see text for details.

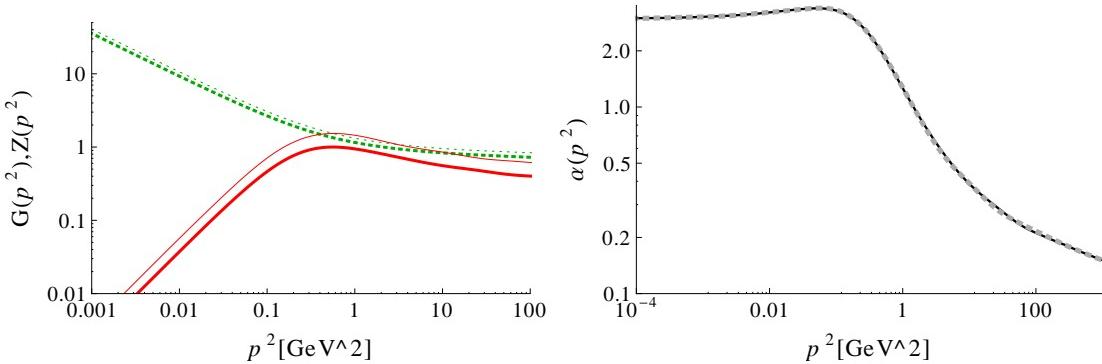


Figure 9: *Left:* Ghost and gluon propagators. The continuous (red) lines corresponds to the gluon, the dashed (green) lines to the ghost. Thick and thin lines corresponds to different choices of $\alpha(\mu^2)$. *Right:* Coupling for the two choices of $\alpha(\mu^2)$. The two lines (black straight and gray dashed) are almost indistinguishable.

for our calculations. The reached precision can be seen from how well the results fulfill the DSEs, i. e., how close $E(p^2)$ approaches zero. Its norm goes with **double** precision down to about 10^{-6} . Using **long double** variables instead this value can be made even lower.

We also tried a second choice for $\alpha(\mu^2)$ to test the code, namely $\alpha(\mu^2) = 0.5$. As expected the propagators change by a constant factor due to multiplicative renormalizability, whereas the running coupling $\alpha(p^2)$ is independent of μ^2 , see, for example, [12, 55]. A comparison between $\alpha(\mu^2) = 1$ and 0.5 is depicted in Fig. 9.

Finally we want to make some technical remarks: With these examples we only want to illustrate the basic use of *CrasyDSE* so the code is not optimized and we expect that the runtime can be improved considerably. Another point is that we also tried a simple fixed point iteration but did not get a solution. This may indicate that this method is not suited for this problem.

7. Summary and outlook

The strength of the framework provided by *CrasyDSE* lies in its ability to handle large numbers of dressing functions. Thus one of its fields of application is the extension of current truncation schemes by including higher vertices and/or enlarging the tensor bases of Green functions. For example, it is expected that going beyond the

interpolations:	linear
expansions:	Chebyshev
solution methods:	fixed point iteration, Newton
quadratures:	Gauss-Legendre, Chebyshev, Fejer, double exponential

Table 4: Currently implemented numerical methods.

current truncation schemes in the Landau gauge makes DSE solutions more competitive to lattice solutions in the mid-momentum regime. Since the number of dressing functions increases at non-zero temperature and/or non-zero density corresponding calculations can profit from *CrasyDSE* too. Finally there are also interesting cases for which no numerical calculations have been done successfully yet because their systems of DSEs are very complex.

Combining *CrasyDSE* with *DoFun* there exists a sound framework for the treatment of all such systems of DSEs, from their derivations to their numeric solutions. Furthermore, we plan to extend *CrasyDSE* by adding further approximation methods or new solving algorithms to the ones given in table 4 as required by individual cases.

Acknowledgments

We would like to thank Reinhard Alkofer, Christian S. Fischer, Markus Hopfer, Axel Maas and Lorenz von Smekal for valuable discussions. We are grateful to Axel Maas for a critical reading of an early version of this manuscript and to Christian S. Fischer for providing numerical results for comparison. Furthermore we want to thank Tina Katharina Herbst for a careful reading of the manuscript and useful comments. MQH is supported by the Alexander von Humboldt foundation. MM acknowledges support by the Doktoratskolleg "Hadrons in Vacuum, Nuclei and Stars" of the Austrian science fund (FWF) under contract W1203-N16.

Appendix A. Initializing and accessing `x_A` and `A`

Here we provide some details on accessing and initializing the arrays `double *x_A` and `double *A` which are members of the structure `struct dse`. Details on their structure can be found as comments in `DSE.cpp`. Here we only describe the functions to access them properly which should be sufficient for most applications. For every DSE we also have to define the number of external momenta and the number of dressing functions, `int dim_x` and `int dim_A`, respectively. For every external momentum the number of interpolation points or expansion coefficients has to be specified. The corresponding values form the array `int *n_A`. As an introductory example for the use of these variables one can consider the interpolation example `interp_main.cpp`.

Appendix A.1. The array `x_A`

The array `x_A` contains the interpolation points and/or Chebyshev nodes depending on the chosen interpolation method. In the case of Chebyshev interpolation only the discrete variables need to be initialized by hand in `x_A`. However when using the DSE solving routines it is necessary that also the continuous Chebyshev

nodes are stored in `x_A`. These are automatically initialized in `x_A` by calling the function `void Cheb_init_cont_xA(void *dse_param)`. In the provided examples `Cheb_init_cont_xA` is called in the member `void (*init_dress)` of the structure `dse`, e. g., `scalprop_init_dress_cheb` for the scalar propagator. Note that due to technical reasons the values for each continuous variable are ordered from low to high for a linear interpolation, but from high to low for a Chebyshev expansion.

Independent of the interpolation method the discrete external momenta have to be initialized by the user in `x_A`. In the case of linear interpolation this is also true for the continuous arguments⁷. Assume we are interested in the grid point `i`. Its grid coordinates are stored in the array `*ind` by calling the function `void index(int *ind, int *n, int dim, int i)`, where `*n` are the number of grid points in every external momentum, given by `n_A`, and `dim` is the number of external momenta, given by `dim_x`. Then we can obtain the index of the `idir`-th component of the `i`-th grid point with the function `int xA_index(int *ind, int idir, void *dse_param)`. When `x_A` is correctly initialized access to the coordinates of the `i`-th grid point is provided via the function `void outer_argument(int i, void *dse_param)` which stores them in `outer_arguments`.

Appendix A.2. The array A for linear interpolation

When linear interpolation is used the array `A` contains the function values at the external momenta `x_A`. The function `int A_index(int *ind, int idress, void *dse_param)` returns the index of the grid point with the coordinates `ind` obtained with the help of the function `index`, see section Appendix A.1. The argument `idress` is the index of the dressing function.

Appendix A.3. The array A for Chebyshev expansion

For a Chebyshev expansion the array `A` contains the Chebyshev coefficients. They can be initialized from a (user-provided) function `func` by calling `void Cheb_init_coeff_mult(double (*func)(double *x, void *param), void (*traf)(double *x, void *param), int dim, int *n_f, int *n_c, double *c_ar, void *param)`. The argument `traf` defines the transformation of the domain of the Chebyshev polynomials ($[-1, 1]$ and direct products thereof) to whatever is the domain of the function. We recommend the function `void Cheb_gen_traf(double *x, void *param)` for this purpose which transforms the interval depending on `int *cheb_trafo`, a member of the corresponding `dse` structure. If `cheb_trafo[i]` is 0/1/2 no transformation/a linear transformation/a logarithmic transformation is employed for the `i`-th external momentum. In what follows we give the expressions usually used as arguments of `Cheb_init_coeff_mult` in parenthesis. `dim` defines the number of continuous arguments (`dim_x - dim_mat`), `*n_f` (`n_A`) is the number of points used for evaluating the inner product to calculate the `*n_c` (`n_A`) Chebyshev coefficients stored in `*c_ar` (`A`) and `param` are some parameters (`dse_param`). In general `n_f=n_c` should be used. We note here that if more than one dressing function is to be interpolated the Chebyshev coefficients can be stored in one array `A` by simply passing the argument `A+idress*n_tot_A` as `*c_ar` where `idress` denotes the `idress`-th function.

⁷For Chebyshev interpolation this has to be done manually as well if one wants other external momenta than the Chebyshev nodes initialized by `Cheb_init_cont_xA`.

Appendix B. Structure of user-defined functions

For the following members of `dse` or `quad` structures the user has to provide functions:

- `void def_domain(double *x, void *dse_param)`: Here the user defines the boundaries of the interpolation domain for the dressing functions. The bounds are saved in the array `double *domain`, a member of the structure `dse_param`. The lower and upper bounds of the i -th external momentum are saved in `domain[2*i]` and `domain[2*i+1]`, respectively. The array `x` refers to the external variables. As examples consider the following functions: `scalgluevert_def_domain` in `scalar_QCD.cpp` or `interp_def_domain` in `interp.cpp`. `def_domain` is a member of a `dse` structure.
- `double interp_offdomain(int *pos, double *x, int iA, void *dse_param)`: This function is required for extrapolation, i. e., when the domain of the interpolation of a dressing function is left. The array `pos` of length `dim_x` contains the values 0, 1 and 2 for every external momentum. If `pos[i]=0`, the interpolation domain for the i -th external momentum is not left, while a value of 1 or 2 means that the lower or upper bounds are crossed, respectively. `pos` is created automatically based on the information provided in `def_domain`. `iA` tells which of the `dim_A` dressing functions is required and the array `x` contains the values of the external momenta. `dse_param` refers to the `dse` structure to which this dressing function belongs. Examples are the functions `ghprop_interp_offdomain_cheb` in `YM4d.cpp` or `scalvert_interp_offdomain` in `scalar_QCD.cpp`. `interp_offdomain` is a member of a `dse` structure.
- `void (*renorm)(void *dse_param)`: This function has to perform several tasks. It does not only implement the renormalization procedure but also contains all steps required to proceed from the results of the integration to the expressions required by the solving algorithms.

The self-energy contributions of single diagrams are calculated by the solving algorithms with the function `selfenergy`. It stores the results for each dressing function and diagram in the `dse` member `double *self_A`. This array is organized as follows: For every quadrature `Q[i]` a block of size `n_loop[i]*dim_A*n_tot_A` is used. These blocks are separated into `n_tot_A` sub-blocks of size `dim_A*n_loop[i]`, i. e., each entry contains the result of the integration of one diagram for a specific external momentum. Appropriately summed up we obtain the result for the right-hand side of a DSE for all external momenta. This summation as well as the renormalization have to be done in `renorm`. Furthermore, if the DSE is not projected directly onto its dressing functions, the linear system of equations to obtain them has to be solved here. Finally, depending on the solving algorithm for the DSEs, either the `dse` members `A` or `E` have to be set: In the case of a fixed point iteration the results can be directly saved to `A` if a linear interpolation is used. For a Chebyshev expansion the new coefficients `A` are calculated with `Cheb_coeff_mult`. If the solving algorithm is Newton's method the renormalization function calculates $E^{(i,k)}$ of Eq. (6). Examples are `scalgluevert_renorm` in `scalar_QCD.cpp`

and `prop_renorm_cheb_secant` in `YM4d.cpp`. `renorm` is a member of a `dse` structure.

- `void integrand(double *erg, double *x, void *int_param)`: The actual kernels for the integration are contained in this function. Every `quad` structure handles the integration of one or several integrals and the values of the integrands at the integration variables given by the array `x` are stored in the array `erg`. The external momenta are accessed via `int_param`, which refers to a `dse` structure. Its member `double *outer_arguments` has to contain the external momenta. `integrand` can be a user-written function, but more commonly it will be created by the *Mathematica* functions of *CrasyDSE*. Examples are `sphere_integrand` in `sphere.cpp`, which was created manually and is thus relatively simple, and `kernelsVertex` in `kernelsAll.cpp` of the scalar QCD example, which was created in `CrasyDSE_YM+scalar.nb`. `integrand` is a member of a `quad` structure.
- `double jacob(double *x)`: This function can be used for the Jacobian of the integral measure. Since the Jacobian is automatically taken into account for every integration, it must always be defined. Thus, if it is already contained in the `integrand`, this function must return 1. The array `x` holds the integration variables. Examples are `sphere_jacob` in `sphere.cpp` and `ghprop_jacob` in `YM4d.cpp`. `jacob` is a member of a `quad` structure.
- `void coeff(double *coefficients, void *int_param)`: Any trivial coefficients of the integrals which do not depend on any momenta can be put into this function. Their values are saved in the array `coefficients` which is multiplied with the results from the integration. `int_param` refers to the `quad` structure of the integration. Similar to `integrand` this function can be written manually or created with *Mathematica*. Examples are `sphere_coeff` in `sphere.cpp`, which was created manually, and `coeffsVertex` in `kernelsAll.cpp` of the scalar QCD example, which was created in `CrasyDSE_YM+scalar.nb`. `coeff` is a member of a `quad` structure.
- `void boundary(double *bound, double *x, int idim, void *int_param)`: The limits of the integration are defined in this function. For the `idim`-th integration variable `bound[0]` and `bound[1]` are set to the lower and upper integration bounds, respectively. The array `x` contains the external momenta, on which the integration bounds may depend. If this is the case, the variable `bound_type` has to be set to 1, see section 4.2. `int_param` refers to the `quad` structure of the integration. Examples are `sphere_boundary` in `sphere.cpp` and `scalgluevert_boundary` in `scalar_QCD.cpp`. `boundary` is a member of a `quad` structure.

Appendix C. Overview of all *C++* functions

This appendix provides tables with all *C++* functions of *CrasyDSE* relevant for the user. Table C.5 contains the most prominent functions of *CrasyDSE*, table C.6 the functions and variables for approximating the dressing functions, table C.7 the functions and variables for the integration and table C.8 further functions and variables of `dse` structures. Details on the functions' arguments are provided in the

code in the function descriptions. For the user's convenience we provide the file *template_DSE.cpp* in the directory *main* which contains a list of all functions and variables that have to be defined for a *dse* structure.

Table C.5: Overview of the main *C++* functions of *CrasyDSE*.

Function	File	Short description
<code>int A_index (int *ind, int dress, void *dse_param)</code>	<i>DSE.cpp</i>	index of coefficient with coordinates <i>ind</i> of <i>dress</i> -th dressing function
<code>void Cheb_coeff_mult (int dim, int *n_f, int *n_c, double *f_ar, double *c_ar)</code>	<i>dressing.cpp</i>	Chebyshev coefficients from array of function values at Chebyshev nodes
<code>double Cheb_gen_dress_interp (double *x, int i, void *dse_param)</code>	<i>dressing.cpp</i>	interpolating Chebyshev polynomial
<code>void Cheb_gen_traf (double *x, void *param)</code>	<i>dressing.cpp</i>	maps domain of Chebyshev polynomials to domain of function
<code>void Cheb_init_coeff_mult (double (*func)(double *x, void *param), void (*traj)(double *x, void *param), int dim, int *n_f, int *n_c, double *c_ar, void *param)</code>	<i>dressing.cpp</i>	Chebyshev coefficients from function
<code>void Cheb_init_cont_xA (void *dse_param)</code>	<i>dressing.cpp</i>	initializes the continuous external variables to the Chebyshev nodes
<code>void dealloc_A_xA (void *dse_param)</code>	<i>DSE.cpp</i>	deallocate members of <i>dse</i> structure
<code>void deallocate_quad (void *quad_param)</code>	<i>quadrature.cpp</i>	deallocate members of <i>quad</i> structure
<code>void get_nw (double *x, double *w, int i, int idim, void *bound_params, void *quad_param)</code>	<i>quadrature.cpp</i>	transformed nodes and weights
Function	File	Short description

Function	File	Short description
void index (int *i, int *n, int dim, int index)	<i>function.cpp</i>	coordinates of index-th entry of a one-dimensional array in a $n[0] \times \dots \times n[dim-1]$ grid
void init_A_xA (void *dse_param)	<i>DSE.cpp</i>	allocate members of <code>dse</code> structure
void init_dse_default (void *dse_param)	<i>DSE.cpp</i>	default values of parameters for allocation of <code>dse</code> structure
void init_quad (void *quad_param)	<i>quadrature.cpp</i>	allocate members of <code>quad</code> structure
void integrate (double *erg, void *quad_param, void *int_param)	<i>quadrature.cpp</i>	integrate a set of functions
double Lin_gen_dress_interp (double *x, int i, void *dse_param)	<i>dressing.cpp</i>	linear interpolation of array of function values
void meta_solve_iter (struct dse *DSE, int ndse, double epsabsstop, double epsrelstop, int iterstop, int output)	<i>DSE.cpp</i>	solves coupled set of DSEs by iteration
void outer_argument (int i, void *dse_param)	<i>DSE.cpp</i>	sets <code>outer_arguments</code> to the i -th external momenta
void plot_credits()	<i>DSE.cpp</i>	prints version and author information
void plot_dressing (void *dse_param)	<i>dressing.cpp</i>	output of dressing functions and interpolation points on screen
void solve_iter (void *dse_param, int output)	<i>DSE.cpp</i>	solves one DSE by iteration
void solve_iter_secant (struct dse *DSE, int ndse, int maxiter, double eps_E, int output)	<i>DSE.cpp</i>	solves array of DSEs with Newton's method
Function	File	Short description

Function	File	Short description
void write_dressing (void *dse_param, char *name, double *addpara, int length)	<i>dressing.cpp</i>	output of dressing functions and interpolation points in file
int xA_index (int *ind, int idir, void *dse_param)	<i>DSE.cpp</i>	index of <i>idir</i> -th direction of a grid point with coordi- nates <i>ind</i>
Function	File	Short description

Table C.6: User defined members of a `dse` structure relevant for the approximation of functions. References to the scalar-gauge field vertex refer to the example of section 5, those to the ghost-gluon system to the example of section 6.

Member	General purpose	DSE usage/example
double *A	interpolation coefficients	e.g., coefficients of A_{p_1} and A_{p_2} for the scalar-gauge field vertex
int *cheb_func_trafo	defines transformation of approximated function for Chebyshev expansion; 0: none, 1: logarithmic	e.g., approximation of the exponential of the ghost dressing function instead of the dressing function itself
int *cheb_trafo	defines transformation function to interval $[-1, 1]$ for Chebyshev expansion; 0: none, 1: linear, 2: logarithmic	e.g., logarithmic transformation of definition interval of ghost dressing function
double *cheb_x	necessary for Chebyshev interpolation with <code>dim_mat</code> >0	see <i>interp.cpp</i> for correct usage
void (*def_domain) (double *x, void *dse_param)	defines domain of definition	e.g., $[0, (\Lambda/2)^2]^2 \times [-1, 1]$ for scalar-gauge field vertex
int dim_A	number of interpolated functions	number of dressing functions for a given Green function, e.g., 2 (A_{p_1} and A_{p_2}) for the scalar-gauge field vertex
int dim_mat	number of discrete arguments of interpolated functions	number of independent Matsubara frequencies
Member	General purpose	DSE usage/example

Member	General purpose	DSE usage/example
<code>int dim_x</code>	number of arguments of interpolated functions	number of arguments of dressing functions, e.g., 3 (p_1^2, p_2^2 and z) for the scalar-gauge field vertex
<code>double (*dress)(double *x, int i, void *dse_param)</code>	returns value of the i -th dressing function at the momenta x	e.g., A_{p_1} and A_{p_2} for the scalar-gauge field vertex
<code>double *E</code>	array of the values on the left-hand side of Eq. (6) for Newton's method	has to be calculated in the renormalization procedure defined by the user, e.g., in <code>prop_renorm_cheb_secant</code> for the ghost-gluon system
<code>void (*init_func)(double *x, void *dse_param)</code>	function used for initialization of the Chebyshev coefficients	ansatz for dressing function with the transformation from <code>cheb_func_trafo</code> taken into account, e.g., ghost dressing function <code>ansatz</code> given by <code>ansatzGh4dLog</code>
<code>double (*interp_offdomain)(int *pos, double *x, int iA, void *dse_param)</code>	is called if <code>dress</code> is evaluated outside domain	e.g., $A_{p_1, p_2} = \tilde{Z}_1$ for $p_1, p_2 \geq (\Lambda/2)^2$ in scalar-gauge field vertex
<code>int *n_A</code>	numbers of expansion coefficients in the <code>dim_x</code> variables	numbers of expansion coefficients for every external momentum
<code>double *x_A</code>	(discrete) interpolation points	external momenta (Matsubara frequencies)
Member	General purpose	DSE usage/example

Table C.7: User defined members of a `quad` structure.
 References to the scalar-gauge field vertex refer to the example of section 5, those to the ghost-gluon system to the example of section 6.

Member	General purpose	DSE usage/example
<code>int bound_type</code>	defines if the integration boundaries depend on the <code>nint para</code> sets of parameters	defines if the integration boundaries depend on the external momenta
Member	General purpose	DSE usage/example

Member	General purpose	DSE usage/example
<code>void (*boundary)(double *bound, double *x, int idim, void *int_param)</code>	defines the integration boundaries for the different integration regions	e.g., for the scalar-gauge field vertex $qs \in [0, \Lambda^2]$, $ct1, ct2 \in [-1, 1]$
<code>void (*coeff)(double *coefficients, void *int_param)</code>	constant factor of integrand	constant factors of self-energy kernels
<code>int dim</code>	number of integration variables	number of loop momentum variables, e.g., 3 ($qs, ct1$ and $ct2$) for scalar-gauge field vertex
<code>void (*init_para)(int i, void *int_param)</code>	initializes the <code>nint_para</code> parameter values	automatically set to void <code>outer_argument</code> which initializes the external momenta
<code>void (*integrand)(double *erg, double *x, void *int_param)</code>	the integrand	kernels of the self-energies
<code>double (*jacob)(double *x)</code>	Jacobian	usually set to one
<code>int nint</code>	number of different integrands that are integrated over the same variables	number of dressing functions multiplied by number of graphs with the same integration variables, e.g., $2 * 2 = 4$ for the scalar-gauge field vertex
<code>int nint_para</code>	number of times the integrands are integrated for different parameter values	number of external momenta at which the self-energy is evaluated, i.e., number of interpolation points
<code>int *n_part</code>	number of integration regions for any of the <code>dim</code> integration variables	e.g., the radial integration for the ghost-gluon system is split into two parts
<code>int *nodes_part</code>	number of quadrature nodes for the different integration regions	e.g., the radial integration of the ghost-gluon system
Member	General purpose	DSE usage/example

Member	General purpose	DSE usage/example
<code>double *param</code>	defines additional parameters in a quadrature rule, e.g., for the implemented double exponential	not used in any of the DSE example, correct usage demonstrated in <code>sphere_main.cpp</code>
<code>int *traj</code>	defines the transformation function from $[-1, 1]$ to the true region of integration for different integration regions	e.g., for the scalar-gauge field vertex the <code>qs</code> integration is mapped via a modified logarithmic mapping whereas the integrations in <code>ct1</code> and <code>ct2</code> are not mapped
<code>int *type</code>	quadrature rules for the different integration regions	e.g., the scalar gluon vertex uses Fejer's second rule for the <code>qs</code> integration, Gauss-Chebyshev quadrature for the <code>ct1</code> integration and Gauss-Legendre quadrature for the <code>ct2</code> integration
Member	General purpose	DSE usage/example

Table C.8: User defined members of a `dse` structure relevant for solving DSEs (not already included in table C.6).

Member	Short description
<code>double anom_dim</code>	anomalous dimension of a dressing function; optional
<code>std::string DSE_name</code>	name of the DSE; optional
<code>double epsabs</code>	stopping criterion for iteration: absolute difference of solutions
<code>double epsrel</code>	stopping criterion for iteration: relative difference of solutions
<code>double h</code>	stepsize for Broyden's method; required only for secant method
<code>void (*init_dress)(void *dse_param)</code>	initializes coefficients <code>*n_A</code> and interpolation points, i.e., initial guess for the dressing function and definition of external momenta
<code>void (*init_renormparam)(void *dse_param)</code>	initializes renormalization parameters
<code>int it_counter</code>	counter for iterations; optional
<code>int maxiter</code>	stopping criterion for iteration: maximal iteration number
Member	Short description

Member	Short description
void (*mod)	model parameters, usually the same for all Green functions
int *n_loop	number of diagrams with the same integral, $*n_loop[i]$, $i \in \{0, \dots, n_loopnumber - 1\}$
int n_loopnumber	number of different integrals
int n_otherGF	number of other Green functions contributing to the given DSE
struct dse *otherGF	one <code>dse</code> structure for every other Green function contributing
double *outer_arguments	stores the external momentum for a given index when the function <code>outer_argument</code> is called
struct quad *Q	one <code>quad</code> structure for every of the <code>n_loopnumber</code> different integrals
void (*renorm) (void *dse_param)	function that is called after calculating the loop integrals of the self-energy in iteration based solving routines
int *renorm_i	index of renormalization point
int renorm_init	determines if <code>*Z_renorm</code> , <code>*renorm_param</code> , <code>*renorm_x</code> and <code>*renorm_i</code> are allocated automatically by <code>init_AxA</code>
int renorm_n_param	number of renormalization parameters
int renorm_n_Z	number of renormalization constants
double *renorm_param	stores renormalization parameters
double *renorm_x	renormalization point
double *Z_renorm	stores renormalization constants
Member	Short description

References

- [1] J. Berges, N. Tetradis, and C. Wetterich, *Phys. Rept.* **363** (2002) 223–386, [arXiv:hep-ph/0005122](https://arxiv.org/abs/hep-ph/0005122).
- [2] J. M. Pawłowski, *Annals Phys.* **322** (2007) 2831–2915, [arXiv:hep-th/0512261](https://arxiv.org/abs/hep-th/0512261).
- [3] H. Gies, [arXiv:hep-ph/0611146](https://arxiv.org/abs/hep-ph/0611146). Presented at ECT* School on Renormalization Group and Effective Field Theory Approaches to Many-Body Systems, Trento, Italy, 27 Feb - 10 Mar 2006.
- [4] O. J. Rosten, *Phys.Rept.* **511** (2012) 177–272, [arXiv:1003.1366 \[hep-th\]](https://arxiv.org/abs/1003.1366).

- [5] R. Alkofer and L. von Smekal, *Phys. Rept.* **353** (2001) 281, [arXiv:hep-ph/0007355](#).
- [6] R. Alkofer, M. Q. Huber, and K. Schwenzer, *Comput. Phys. Commun.* **180** (2009) 965–976, [arXiv:0808.2939 \[hep-th\]](#).
- [7] D. Binosi and J. Papavassiliou, *Phys. Rept.* **479** (2009) 1–152, [arXiv:0909.2536 \[hep-ph\]](#).
- [8] C. D. Roberts, [arXiv:1203.5341 \[nucl-th\]](#).
- [9] J. Berges, *Phys. Rev.* **D70** (2004) 105010, [arXiv:hep-ph/0401172](#).
- [10] L. von Smekal, A. Hauck, and R. Alkofer, *Ann. Phys.* **267** (1998) 1, [arXiv:hep-ph/9707327](#).
- [11] C. S. Fischer and R. Alkofer, *Phys. Rev.* **D67** (2003) 094020, [arXiv:hep-ph/0301094](#).
- [12] C. S. Fischer, A. Maas, and J. M. Pawłowski, *Annals Phys.* **324** (2009) 2408–2437, [arXiv:0810.1987 \[hep-ph\]](#).
- [13] M. Q. Huber, K. Schwenzer, and R. Alkofer, *Eur. Phys. J.* **C68** (2010) 581–600, [arXiv:0904.1873 \[hep-th\]](#).
- [14] L. von Smekal, R. Alkofer, and A. Hauck, *Phys. Rev. Lett.* **79** (1997) 3591–3594, [arXiv:hep-ph/9705242](#).
- [15] A. Hauck, L. von Smekal, and R. Alkofer, *Comput. Phys. Commun.* **112** (1998) 166, [arXiv:hep-ph/9804376](#).
- [16] S. Wolfram, *The Mathematica Book*. Wolfram Media and Cambridge University Press, 2004.
- [17] M. Q. Huber and J. Braun, *Comput. Phys. Commun.* **183** (2012) 1290–1320, [arXiv:1102.5307 \[hep-th\]](#).
- [18] M. Q. Huber, *On gauge fixing aspects of the infrared behavior of Yang-Mills Green functions*. Springer, 2012. [arXiv:1005.1775](#). Ph.D.Thesis, Karl-Franzens-University Graz, 2010.
- [19] M. Q. Huber, R. Alkofer, and S. P. Sorella, *Phys. Rev.* **D81** (2010) 065003, [arXiv:0910.5604 \[hep-th\]](#).
- [20] M. Q. Huber, R. Alkofer, and S. P. Sorella, *AIP Conf. Proc.* **1343** (2011) 158–160, [arXiv:1010.4802 \[hep-th\]](#).
- [21] M. Q. Huber, R. Alkofer, and K. Schwenzer, *PoS FACESQCD* (2010) 001, [arXiv:1103.0236 \[hep-th\]](#).
- [22] T. Fischbacher and F. Synatschke-Czerwonka, [arXiv:1202.5984 \[physics.comp-ph\]](#).
- [23] N. Bogoliubov and O. Parasiuk, *Acta Math.* **97** (1957) 227–266.

- [24] W. Zimmermann, *Commun.Math.Phys.* **15** (1969) 208–234.
- [25] K. Hepp, *Commun.Math.Phys.* **2** (1966) 301–326.
- [26] J. C. Bloch, [arXiv:hep-ph/0208074 \[hep-ph\]](#). Ph.D. thesis (1995), University of Durham.
- [27] D. Atkinson and J. C. R. Bloch, *Phys. Rev.* **D58** (1998) 094036, [arXiv:hep-ph/9712459](#).
- [28] A. Maas, *Comput. Phys. Commun.* **175** (2006) 167–179, [arXiv:hep-ph/0504110](#).
- [29] D. Horvatic, private communications, 2011.
- [30] M. Mitter, M. Hopfer, B.-J. Schaefer, and R. Alkofer, in preparation.
- [31] C. S. Fischer and R. Alkofer, *Phys. Lett.* **B536** (2002) 177–184, [arXiv:hep-ph/0202202](#).
- [32] J. S. Ball and T.-W. Chiu, *Phys. Rev.* **D22** (1980) 2542.
- [33] P. Maris and C. D. Roberts, *Phys.Rev.* **C56** (1997) 3369–3383, [arXiv:nucl-th/9708029 \[nucl-th\]](#).
- [34] P. Maris and P. C. Tandy, *Phys.Rev.* **C60** (1999) 055214, [arXiv:nucl-th/9905056 \[nucl-th\]](#).
- [35] R. Alkofer, P. Watson, and H. Weigel, *Phys.Rev.* **D65** (2002) 094026, [arXiv:hep-ph/0202053 \[hep-ph\]](#).
- [36] C. S. Fischer and J. A. Müller, *Phys. Rev.* **D80** (2009) 074029, [arXiv:0908.0007 \[hep-ph\]](#).
- [37] R. Alkofer, C. S. Fischer, F. J. Llanes-Estrada, and K. Schwenzer, *Annals Phys.* **324** (2009) 106–172, [arXiv:0804.3042 \[hep-ph\]](#).
- [38] L. Fister, R. Alkofer, and K. Schwenzer, *Phys. Lett.* **B688** (2010) 237–243, [arXiv:1003.1668 \[hep-th\]](#).
- [39] R. Alkofer, L. Fister, A. Maas, and V. Macher, *AIP Conf.Proc.* **1343** (2011) 179–181, [arXiv:1011.5831 \[hep-ph\]](#).
- [40] V. Macher, A. Maas, and R. Alkofer, [arXiv:1106.5381 \[hep-ph\]](#).
- [41] M. Hopfer, diploma thesis (2011), Karl-Franzens-University Graz.
- [42] A. Maas, *PoS FACESQCD* (2010) 033, [arXiv:1102.0901 \[hep-lat\]](#).
- [43] W. J. Marciano and H. Pagels, *Phys. Rept.* **36** (1978) 137.
- [44] A. Aguilar, D. Binosi, and J. Papavassiliou, *Phys.Rev.* **D78** (2008) 025010, [arXiv:0802.1870 \[hep-ph\]](#).
- [45] C. S. Fischer, [arXiv:hep-ph/0304233 \[hep-ph\]](#). Ph.D. thesis, Eberhard-Karls-Universität zu Tübingen (2003).

- [46] C. S. Fischer and J. M. Pawłowski, *Phys. Rev.* **D80** (2009) 025023, [arXiv:0903.2193 \[hep-th\]](#).
- [47] A. Cucchieri, A. Maas, and T. Mendes, *Phys. Rev.* **D77** (2008) 094510, [arXiv:0803.1798 \[hep-lat\]](#).
- [48] C. Lerche and L. von Smekal, *Phys. Rev.* **D65** (2002) 125006, [arXiv:hep-ph/0202194](#).
- [49] W. Schleifenbaum, A. Maas, J. Wambach, and R. Alkofer, *Phys. Rev.* **D72** (2005) 014017, [hep-ph/0411052](#).
- [50] M. Pennington and D. Wilson, *Phys. Rev.* **D84** (2011) 119901, [arXiv:1109.2117 \[hep-ph\]](#).
- [51] P. Boucaud, J.-P. Leroy, A. L. Yaouanc, J. Micheli, O. Pene, and J. Rodriguez-Quintero, *JHEP* **0806** (2008) 012, [arXiv:0801.2721 \[hep-ph\]](#).
- [52] D. Zwanziger, *Phys. Rev.* **D65** (2002) 094039, [arXiv:hep-th/0109224](#).
- [53] R. Alkofer, C. Fischer, and L. von Smekal, *Acta Phys.Slov.* **52** (2002) 191, [arXiv:hep-ph/0205125 \[hep-ph\]](#).
- [54] L. von Smekal, K. Maltman, and A. Sternbeck, *Phys.Lett.* **B681** (2009) 336–342, [arXiv:0903.1696 \[hep-ph\]](#).
- [55] R. Alkofer, C. S. Fischer, and F. J. Llanes-Estrada, *Phys. Lett.* **B611** (2005) 279–288, [arXiv:hep-th/0412330](#).